1990

The object-oriented design of a hardware description language analyser for the DIADES silicon compiler system

Lian Yang
Portland State University

AN ABSTRACT OF THE THESIS OF Lian Yang for the Master of Science in Electrical and Computer Engineering presented December 7, 1990.

Title: The Object-oriented Design of a Hardware Description Language Analyser for the DIADES Silicon Compiler System

APPROVED BY THE MEMBERS OF THE THESIS COMMITTEE:

[redacted]

Marek A. Perkowski, Chairman

[redacted]

Michael A. Driscoll

[redacted]

Harry Porter

This thesis is one of the first to introduce a systematic and general Source Language Analysis System (called SLA) for a high-level synthesis system.

A new methodology in the SLA design is reported. This methodology is realized by employing the object-oriented design technique, which is based on a data model called *object* and the *object Directed Acyclic Graph (DAG)*. This methodology has shown advantages over the traditional top-down design methodology by its power of modeling the design entities of a high-level synthesis system and by its extensibility.

A set of formal expressions and rules for applying object-oriented design to the high-level synthesis area is also introduced in the thesis. This set includes the formal

definition of object, object transformations, and the relationships between objects.

Because of the author's contribution, the DIADES system has improved its performance in two aspects:

1. Better data modeling: the Object DAG integrates levels of design information.

2. Greater extensibility: system level and user-controlled extension are realized.

THE OBJECT-ORIENTED DESIGN OF A HARDWARE DESCRIPTION LANGUAGE ANALYSER FOR THE DIADES SILICON COMPILER SYSTEM

by

LIAN YANG

A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
ELECTRICAL AND COMPUTER ENGINEERING

Portland State University
1991

TO THE OFFICE OF GRADUATE STUDIES:

The members of the Committee approve the thesis of Lian Yang presented December 7, 1990.

[redacted]

Marek A. Perkowski, Chairman

[redacted]

Michael A. Driscoll

[redacted]

Harry Porter

APPROVED:

[redacted]

Rolf Schaumann, Chairman, Department of Electrical Engineering

[redacted]

C. William Savery, Interim Vice Provost for Graduate Studies and Research

ACKNOWLEDGEMENT

Many people have given me support, advice, and all kinds of helps during my two years of graduate study at PSU. In particular, my advisor Dr. Perkowski have been giving me extensive advice and very patient with me through my entire study process. The experience of studying and working with him is precious and unforgettable. Special thanks are due to my uncle Francis C. Young, who provided the initial financial support for me. Without his generous help, it would have been impossible for me to come to study in the States.

I would like to thank David Smith, whose support and cooperation in various forms have helped me finish the project and write the thesis. I owe my gratitude to the following people who have provided valuable comments on my thesis: Dr. Harry Porter, Dr. Michael Driscoll, and Ms. Ellen M. Dull.

I would like to mention Ms. Shirley Clark particularly for her kindness and helpfulness to me. I would also extend my thanks to J. Liu, S. Zhang, J. Li, and all my Chinese folks, who have given me tremendous encouragement and support.

Portland, Oregon Lian Yang

TABLE OF CONTENTS

CHAPTER PAGE

CHAPTER PAGE

LIST OF FIGURES

GLOSSARY AND NOTATIONS

AEF - ADL Extension Format

AOS - ADL Object Stack

DDS - DIADES Digital System

DAG - Directed Acyclic Graph

HDL - Hardware Description Language

HBDL - Hardware Behavior Description Language

IR - Intermediate Representation

OOP - Object-oriented Programming

PCG - P-graph Code Generator

SLA - Source Language Analyser

TAG90 - ADL Analysis Program (ADL Compiler)

RT - Register Transfer

Some Notations Used in the Thesis

C-object - an instance of class C

<i C n> - a C-object with an identifier *i* and a state *n*

S(o) - The set of states of an object o

S(A) - The set of states of an A-object

$A \subset B$ - A inherit_from B

$A << C$ - C is composed_of A

$B < A$ - B is supervised_by A

CHAPTER I

INTRODUCTION

1.1 OVERVIEW

As the VLSI design process becomes increasingly dependent on automation tools, various *high-level synthesis systems,* such as CMU-DA [Park 1988, 1986; Hafe 1982], MIMOLA [Marw 1979], and McPitts [Sout 1983] have emerged. A high-level synthesis system takes as input an abstract *behavioral description* of the system to be designed and, step by step, adds implementation information until a physical structure implementing the described behavior has been fully specified [McFa 1983].

High-level synthesis systems play an active role in *silicon compiler systems.* In a silicon compiler system a *hardware description language (HDL)* is the source language. The output is a circuit design in an appropriate specification [Aho 1989].

> [Definition1.1] A high level behavioral description language (HBDL) is a HDL that describes only the functional behavior of a digital circuit.

In [Aylo 1986] eight HBDLs are analysed and compared. A high-level synthesis system can be viewed as the supporting environment of a HBDL. It eventually compiles the HBDL into a hardware circuit, which has the behavior equivalent to that specified by the source HBDL description.

The *DIADES* system is a high-level synthesis system developed at the Department of Electrical Engineering, Portland State University. *ADL (Automated Design Language)* is the HBDL of the DIADES system. In other words, the DIADES system is

the ADL supporting environment. The structure of the DIADES system is shown in Figure 1.

In DIADES, ADL is translated into an internal Lisp-based description form called Program Graph (P-graph) by a program called TAG90. In this thesis, the function played by TAG90 is called the "ADL Analyser." Generally, each high-level synthesis system has a software module that translates its source HBDL into an *intermediate representation (IR)*. In this thesis the software module in an automated synthesis system is called the Source Language Analyser (SLA).

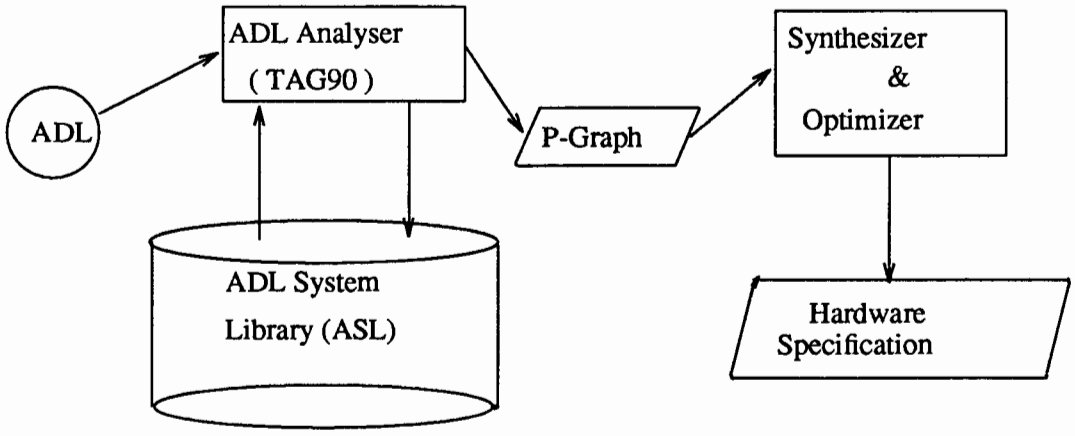


Figure 1. The DIADES system.

1.2 THE PURPOSE OF THE THESIS

1.2.1 Systematic Development of the SLA

This thesis is one of the first in the literature to introduce the systematic and general SLA development method. The SLA is an important part in the HBDL support environment. An SLA analyses the hardware descriptions for static errors and translates HBDL text to an IR. The IR can then be transformed into the *data path (DP)* and *control unit (CU)* specifications through a resource allocation and scheduling process [Sun 1988;

Park 1986, 1988; Paul 1989], and a CU synthesis process [Liu 1989; Yang 1988]; therefore, the SLA can be regarded as the first pass of a high-level synthesis system.

[Farr 1989] describes an HDL compiler based on the compiler construction tool called Attribute Grammar; however, this paper describes few issues related to the high-level synthesis process and the system that it supports is not a high-level synthesis system. On the other hand, most of the well known references in the high-level synthesis area [McFa 1990-b; Lips 1986; Hafe 1982; Kowa 1985; Marw 1979; Park 1986, 1988; Pang 1987] do not address the issues related to the SLA design. The main reason for this seems to be that the traditional top-down [Park 1988, 1986; Marw 1979] and bottom-up [Sout 1983] approaches in high-level synthesis need only very simple, straightforward SLAs and there is no need to address the SLA design issues particularly.

This thesis is believed to be the first (or one of the first) in the high-level synthesis literature that addresses in detail the entire process of designing an SLA and especially the *object-oriented design* (will be briefly introduced in section 1.2.3 and discussed in details in Chapter IV) issues. The SLA of the DIADES system (i.e., the ADL Analyser called TAG90) is presented and special issues involving SLA design are addressed. These special issues include:

- HBDL's purpose, functionality, supporting tools, and its extensibility.
- The intermediate notation -- the target code of SLA, its flexibility, versatility, and extensibility. These are important issues related to the efficiency of the entire HBDL support environment.
- The exploration of new methodologies in high-level synthesis area. This thesis will discuss applying object-oriented design methodology to high-level synthesis.

1.2.2 A New Methodology in High-level Synthesis Area

Most systems in the high-level synthesis area have been using a purely top-down design methodology [McFa 1990-b]. They begin by translating the behavior description into a data flow graph followed by a series of transformations which are applied to make the design more efficient [Tric 1985]. This approach lacks the ability to use the detailed low-level physical information in choosing a *register transfer (RT)* structure [McFa 1990-b]. It also lacks extensibility, an important feature for a HBDL [Aylo 1986]. The advantages of using object-oriented design techniques within the VLSI area have been discussed in [Wolf 1986; Ayer 1989; Gupt 1989]. There even exists a hardware description language that is object-oriented [Sugi 1986]. However, all of them use the object-oriented data model (data base/library) at the RT level. In [Katz 1987], a CAD design version control system is proposed using the object-oriented design method. The object that describes a design is discussed; however, these objects correspond to the versions of the design results and not to the design process itself. In this thesis the design process of the ADL Analyser employs an object-oriented data model and programming techniques, which has the following advantages over the traditional top-down design methods used in most high-level synthesis systems:

- Integrating levels of design. McFarland [1990-a] addressed that it is important to maintain a single representation containing all levels and domains of design information in order to integrate levels of design. The lack of such kind of single representation is one of the main drawbacks of the current high-level synthesis systems. This thesis reports using the class DAG (which will be addressed in detail in Chapter IV) as such kind of single representation. The author of this thesis considers the class DAG approach to the integration of levels of design as a main contribution to the high-level synthesis methodology.

- Better data modeling. The object-oriented data model employed can simulate

the hardware design entities at various design levels [Ayer 1989], unlike the traditional top-down approaches where the data models are black boxes which are passive, abstract, and lack communication between models [McFa 1989-b].

- More systematic design. Using the object-oriented data model and programming techniques, the design process and the coding process are organized according to requirements of the modern software engineering theory. The gap between the conceptual design and the coding is thus dramatically reduced [Meye 1988].

1.2.3 Formal Models of the Object-oriented Design

As a new contribution to the application of the object-oriented programming to the SLA design, the formal models of ADL objects, the relationships between objects, and the object transformations are introduced in this thesis. The class *Directed Acyclic Graph (DAG)* is also introduced to illustrate the relationships between object classes. These formal models improve the expressionability of the object-oriented design and facilitate the application of the object-oriented programming to the high-level synthesis system. In this thesis the object-oriented design or the object-oriented design methodology refers to the following two aspects of effort:

1. the analysis of the class DAG and the design of the object transformations.
2. the object-oriented programming techniques [Wien 1988] employed in the software design.

The details of the object-oriented design methodology will be addressed in Chapter IV.

1.2.4 A Systematic Approach to the Extensibility of the High-level Synthesis System

Language extensibility of an HBDL is an important feature in designing a high-level synthesis system. In [Aylor 1986], language extensibility is regarded as one of the most important features of a hardware description language and one of the goals of the VHDL language design. There are two main criteria in regard to language extensibility

of an HBDL:

1. The language should be structured to be independent of a specific technology, preferred methodology, or design style.

2. The language should be able to withstand technological advances and provide the facility to incorporate new and innovative design techniques. It might be advisable for some extensions to be under user control.

The features of ADL meet criterion 1, since ADL is a high-level algorithmic hardware behavioral description language. In [Lips 1986], criterion 2 is approached by allowing the user to define his own data types. A different approach is used in this thesis. In Chapter V the language extensibility of ADL is discussed in detail. Two kinds of extension process will be investigated. A user controlled extension of ADL is proposed.

1.3 THE OUTLINE OF THE THESIS

This thesis has the following organization:

• Chapter II will address the TAG90 architecture and analyse each program module of TAG90. Since TAG90 employs the compiler generation tools YACC and LEX, it is built differently from the traditional top-down approach and the design focus has been changed. This chapter provides the basic ideas in regard to how TAG90 is modeled and organized.

• The focus of Chapter III is on the analysis of the salient features of the ADL language and the general architecture underlying the ADL program. The architectural description, which separates the control flow from the dataflow, as well as a formal definition representing the general architecture of DIADES Digital System (DDS) are given. The intermediate representation of ADL and P-graph is then described based on the formal definition.

• Chapter IV, the most crucial part of this thesis, presents the process of analysing the ADL language using object-oriented methodology. Formal models of *object* and *object transformations* are presented. The discussion ends with a comparison between this new methodology and the traditional top-town approach to the SLA design. Using this methodology, the P-graph generation process is centered around the generation and manipulation of the ADL objects. (The concept of the ADL objects will be discussed in Chapter IV.) A set of formal expressions for the object-oriented programming in the SLA application is introduced. The formalization of object-oriented programming is expected to contribute to broader applications of object-oriented design in the high-level synthesis area.

• In Chapter V the extensibility of TAG90 is presented in which two approaches to the system extension are compared and discussed in detail.

• In Chapter VI the programming aspects of TAG90 are presented beginning with a more detailed discussion on the features of YACC and LEX. However, the content of this chapter will center on the P-graph Code Generator (PCG) module, which consists of the semantic routines, the definition of the ADL classes, and the transformations on objects of these classes.

• Chapter VII concludes this thesis with a summary and proposal of future work towards the improvement of TAG90 and the ADL language.

• Appendix A contains a complete example of analysing an ADL program, the ADL program source file *adl.ex*, and an example of running TAG90.

• Appendix B presents the YACC specification file for the ADL syntax.

CHAPTER II

THE ARCHITECTURE OF TAG90

This chapter presents an overview of the program architecture and organization of TAG90. TAG90 employs compiler construction tools and object-oriented design. This approach involves building a compiler by distributing the tasks of the compilation process to three major modules: the scanner, the parser, and the semantic routines. The traditional code generation process is abstracted and hidden from the user. This program structure and organization build the fundamentals of the system's flexibility and extensibility. This organization does not emphasize the functions that generate the P-graph code. Instead, TAG90 is based on the analysis of the underlying features of the digital system, as described by the ADL program, and the generation and manipulation of the data (objects) related to these features. Because of the difference in the emphasis of the program, the syntax of both the source and the target language will be discussed in the next chapter, while the general structure and organization of the program TAG90 is presented in this chapter.

2.1 INTRODUCTION

The major operations in a conventional compiler are illustrated in Figure 2. Compilation begins with a source text file where the compiler first sees a stream of source characters. The character stream is then subdivided into a sequence of *tokens* by a *scanner*. A token may be a single character or a special sequence of characters.

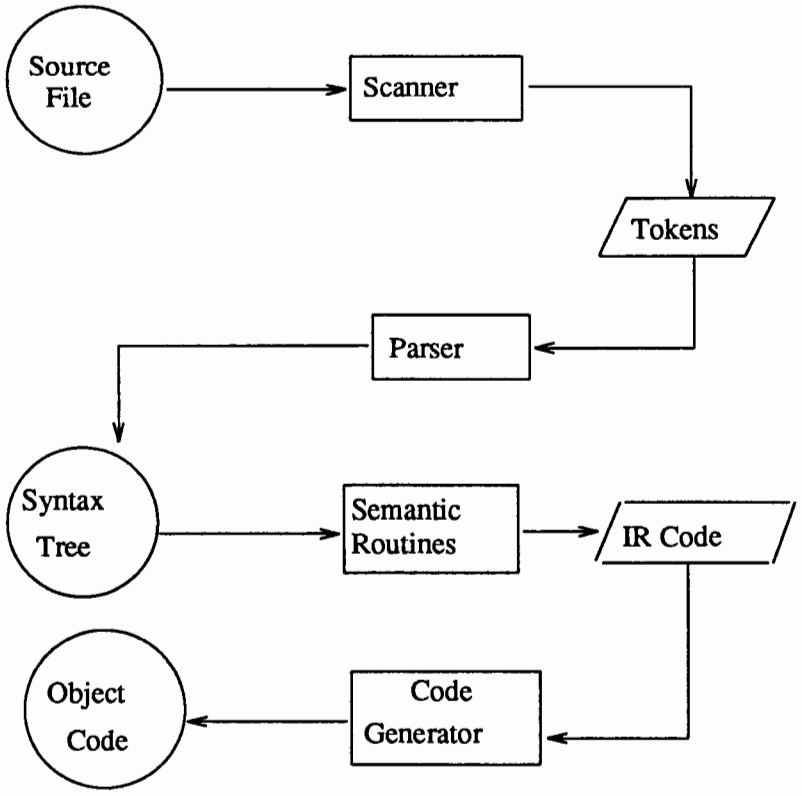


Figure 2. A conventional compilation process.

Examples of tokens in ADL are identifiers (names assigned to variables, statement labels, etc.,) numbers, and special symbols, such as ":=". For example, the following ADL source code

```
Stmt1:  if (a > 1) a := a - 1;
```

would be translated by a scanner into the token sequence:

```
Stmt1:
if
(
a
>
1
)
a
:=
a
-
1
;
```

The token sequence emitted by the scanner is next processed by a *parser,* whose task is to determine the underlying structure or "meaning" of the program. The parser considers the context of each token and classifies groups of tokens into larger entities such as *declarations, statements,* and *control constructs.* While parsing, the parser verifies the correctness of the syntax and, if a syntax error is found, it issues a suitable diagnostic. As syntactic structure is recognized, the parser calls the corresponding *semantic routines.*

Semantic routines perform two functions. They check first the static semantics of each construct, and then they verify that the construct is legal and meaningful. If the construct is semantically correct, semantic routines translate it into an IR. The IR code is then mapped into target machine code by the *code generator.* This requires detailed information about the target machine and often involves a machine-specific optimization.

2.1.1 The TAG90 Structure

Due to the employment of object-oriented design methodology in TAG90, the program structure of TAG90 is different from that of a conventional compiler system, as shown in Figure 2. In TAG90 there is no module called the code generator; instead, a module called PCG is built to generate and manipulate *ADL Objects.* The ADL Objects package together the data and the operations for data manipulation, rather than being passive and containing the data only, as in the traditional IR.

From the functional point of view, the PCG is similar to the combination of the semantic routines and the code generator shown in Figure 2. However, it is different in both concept and implementation. In the PCG design the IR of a conventional compiler is replaced by the ADL Objects; the code generator of a conventional compiler is replaced by some *methods* of the objects themselves. Therefore, the PCG reflects a new approach in the SLA design. Instead of describing the functions to generate the P-graph representation, the PCG design is based on the manipulation of the ADL Objects. (The PCG principle and implementation details will be addressed in Chapter IV.) The general structure of TAG90 is shown in Figure 3.

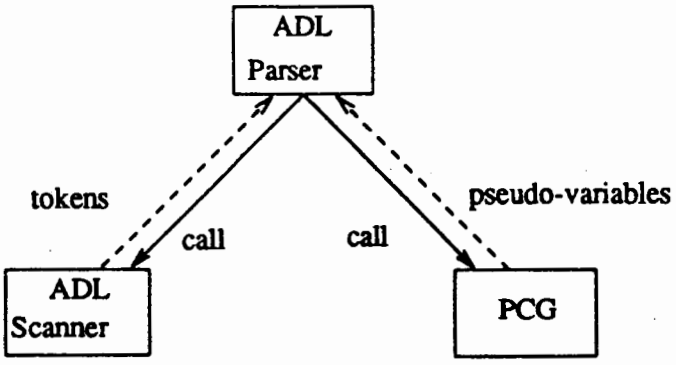


Figure 3. The program of TAG90.

As a program, the main routine of TAG90 is the ADL parser. As illustrated in Figure 3, the relation between the three parts of TAG90 is the following: the parser calls the low-level input routine (the scanner) to pick up the basic items (called tokens) from the input stream (ADL source). These tokens are organized according to the input structure rules, called the grammar rules. When one of the rules has been recognized, the user code supplied for this rule, a semantic routine, is invoked. The PCG contains the set of all semantic routines. The main modules of TAG90 are described respectively in the following sections.

2.2 THE ADL SCANNER

In TAG90 the scanner is constructed by the LEX, the lexical analyzer construction tool developed by M.E.Lesk and E.Schmidt of Bell Laboratories [Schr 1985]. LEX produces an entire scanner module that can be compiled and linked with other compiler modules. LEX reads the lexical specification file and generates the scanner procedure called *yylex()*, which is an integer-valued function. The scanner reads the input ADL source code stream and provides tokens to the parser. The lexical specifications for LEX use regular expressions. For example, in the lexical specification file,

```
digit       [0-9]
name        [a-zA-Z][a-zA-Z_0-9]*
```

specifies the digit and the variable name of the input stream. Program fragments are usually associated with regular expressions. For example,

```
"=="        return token(EQ);
```

associates the C statement "return token(EQ)" with the regular expression "==." When a "==" symbol is seen in an ADL program, a token called EQ is returned to the parser.

2.3 THE ADL PARSER

2.3.1 Introduction

YACC is employed in the TAG90 project to automatically generate an extensible ADL parser. YACC is an LALR(1) (stands for "A 1-symbol Look Ahead Left-to-Right grammar") parser generator developed by S.C. Johnson and others of the Bell laboratories [Schr 1985]. YACC is an acronym for "Yet another compiler-compiler." It provides for semantic stack manipulation and the specification of semantic routines. YACC generates an integrated parser, which is an integer-valued function called *yyparse()*. The parser can call the scanner generated by LEX or a hand-coded scanner written in C.

The input to YACC is a YACC specification file. YACC turns the specification file into a C program, which parses the input ADL source program according to the given specification. Since the algorithm used to transform the specification file into a parser in the C programming language is an automatic process, it will not be discussed here; the emphasis of this thesis is on the specification file itself. A YACC specification file contains three sections: the declarations, the grammar rules, and the programs. The sections are separated by double percent ("%%") marks.

2.3.2 Basic Specifications

A full specification file looks like this:

```
declarations
%%
rules
%%
programs
```

The declaration section is like the declaration section of a C program, which contains the C preprocessor macros, the variable declarations, and the block declaration. The tokens are also declared in this section.

The rule section is made up of one or more grammar rules. Only LALR(1) grammar can be represented by YACC rules. A grammar rule has the form:

```
A : BODY;
```

A represents a nonterminal name and *BODY* represents a sequence of zero or more names and literals. For example, the ADL program can be defined by the rule:

```
program
    : program_head program_body
    ;
```

In the above, *program_head* and *program_body* are nonterminals as well and need further rules to define them. The program section includes some general functions used in the semantic routines. This section is optional.

2.4 ADL SEMANTIC ROUTINES

The PCG is composed of semantic routines. In a YACC specification the user may associate semantic routines with each grammar rule. They are executed whenever the grammar structures specified by the grammar rules are recognized. The routines may return values and may obtain the values returned by previous routines. A semantic routine is also called an *action* and is specified by one or more statements enclosed in curly braces "{" and "}." For example,

```
type
    : INT
          { x = 1; }
    | LOGICAL
          { x = 2; }
    ;
```

is an actual grammar rule with actions in C code.

2.5 THE P-GRAPH GENERATOR

Unlike a conventional compiler, TAG90 does not have a tangible code generator module for the generation of P-graph, because TAG90 employs object-oriented design. Its semantic routines generate ADL Objects instead of an IR. The formation and generation of the object code are performed by certain methods within certain ADL Objects.

At this stage the design approach has been totally changed. The details involved in the PCG design will be addressed in Chapter IV.

2.6 THE CODE ORGANIZATION OF TAG90

The parser program is organized around the YACC specification rules that describe the syntax of ADL. The semantic routines are written in C++ code and can be either in-line expressions, calls to out-of-line, separate functions, or some combination of those. The YACC specification file is the input to YACC, which generates the ADL parser.

The lexical scanner is organized around the regular expressions that describe the lexical rules of the ADL. Some regular expressions have program fragments associated with them. The LEX specification file is the input to LEX, which generates the ADL scanner. The main program contains only the call to the parser called *yyparse()* generated from the YACC specification file. The code organization of TAG90 is shown in Figure 4 (see the next page.) This chapter has outlined the systematic way to realize the SLA of the DIADES system TAG90. By using the compiler construction tools YACC and LEX and object-oriented design methodology, the SLA has some new features, which will be discussed in the following chapters. The ADL language and the P-graph notation will be discussed in the next chapter as the necessary prerequisites for understanding the new contributions.

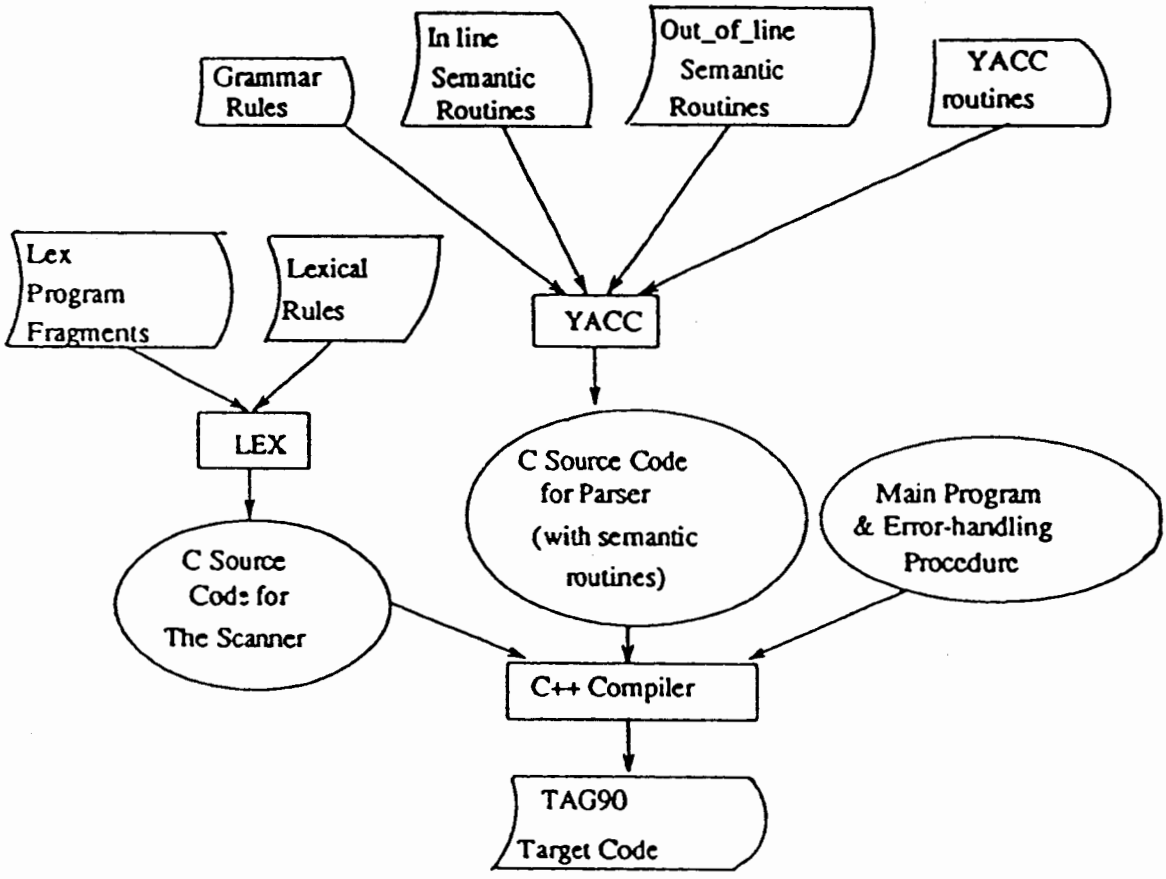


Figure 4. The TAG90 code organization.

CHAPTER III

THE BASIC DATA FORMATS RELATED TO TAG90

This chapter begins with a discussion of the ADL language features with an example of an ADL program. The basis for ADL analysis follows by giving the general abstract architecture of the digital system synthesized by the DIADES system. The architectural description format for the DIADES system is introduced and based on this format, the P-graph is discussed and compared to that of the architectural description format.

3.1 INTRODUCTION TO THE ADL LANGUAGE

The focus of this section is on the major features of ADL. Historically, ADL and its compiler, TAG, had the following drawbacks which inspired this project:

- The old ADL's Lisp-based syntax makes it difficult to read.
- The old ADL compiler has bugs and many of ADL's unique features are not as well implemented as they could be.
- The old ADL was not extensible even though the extensibility of a high-level synthesis system is important.

The syntax of the ADL language has been redefined recently by the author and David Smith. The new ADL keeps the essence of the old ADL which was created by M. Perkowski [Perk 1988], but changes its appearance by adopting a C-like syntax. Adopting the C-like syntax allows the user to master it more easily, as C has been widely used

in the computer industry.

The features of ADL include:

• ADL is an algorithmic language describing the behavior of a digital system. It can support hierarchical design based upon structural hierarchy, behavioral abstraction, or both. The advantage of this approach is that it allows great flexibility in the exploration of possible designs.

• ADL is versatile in its applications since it is not limited to a particular kind of application. Currently, most behavioral description languages are limited to specific applications such as signal processing or microprocessor design [Pang 1987].

• ADL provides various kinds of control constructs and parallel programming facilities [Perk 1989].

• The ADL analyzer uses object-oriented programming methodology and provides for an object-oriented data library called ASL (ADL System Library.) This feature will make ADL one of the most extensible and flexible languages in this area.

3.1.1 ADL Program Structure

An ADL program describing a digital system has four sections. The first section contains a few lines of parameters for the ADL compiler. The second section is the declaration section, which declares variables used in the program and blocks called in the program. Variables represent the inputs and outputs of a system, as well as internal registers or memories. The blocks are like functions of the conventional programming languages and have special hardware implementation. The third section contains the algorithm description. The fourth section contains the block (subroutine) descriptions. The structure of an ADL program is as following:

```
/* Section 1: compiler parameters */
compiler_macros
/* Section 2 : declarations */
ADL program_id program_name
{ Variable declaration section }
{ Block declaration section }
/* Section 3 : algorithm */
start;
{ Algorithm }
end.
/* Section 4: block function description */
{ Block description }
.....
{ Block description }
```

In the above, "/*" and " */" enclose program comments as in the C programming language. The five sections are explained below.

Section 1. Compiler macros are similar to the C preprocessor commands and are used to declare clock, register length, and links to the ADL system library.

Section 2. *Program_id* is the identification for the program. In other words, it represents the identification of the digital circuit described by the program. *Program_name* is the name of the program used to symbolically describe the functionality of the program. The variable declaration section describes the input, output, and internal variables. In the block declaration section the blocks used in the ADL program are declared. For example,

```
Block { b1(int i1, int i2; int o); b2(int i; int o)}
```

declares two blocks b_1, and b_2.

Section 3. The algorithm section describes the behavior of the circuit using ADL statements. In general, there are two kinds of statements: control statements and assignment statements. The main program begins at *start* and ends with *end.*

Section 4. The last section describes the blocks declared in section 3.

3.1.2 An ADL Example

Several examples of ADL programs and the synthesis of their corresponding data path and control unit have been shown in [Liu 1989]. One of them is an ADL program for an eight-instruction CPU. With the new version of ADL, this example looks like this:

```
.de _register 8
.de _clock   1000

adl c 8_instr_cpu;
intern { logical read, write;
         int pc, AR, IR, DR, AC, k, mem[k];
       }
start;
 AR := pc;
 DR := mem [AR];
 sim { pc := pc + 1; IR := DR; }

 if (and (not DR(2))            /* load */
         (not DR(1))
         (not DR(0)))
    {
     AR := DR;
     DR := mem[AR]; AC := DR; }
 else
   if (and (not DR(2))           /* store */
           (not DR(1))
           DR(0) )
     {
      AR := DR;
      DR := AC;
      mem[AR] := DR; }
   else
    if (and (not DR(2))          /* add */
            DR(1)
            (not DR(0)))
       {
        AR := DR;
        DR := mem[AR];
        AC := AC + DR; }
```

```
else
 if (and DR(2)              /* jump */
        (not DR(2))
        (not DR(1))
        (not DR(0)))
   pc := DR;
 else
  if (and DR(2)             /* jumpz */
         (not DR(1))
         DR(0))
    if(AC == 0) pc := DR;
  else
   if (and DR(2)            /* comp */
          DR(1)
          (not DR(2)))
     {
       AC := (not AC);
     }
   else
    if (and DR(2)           /* right shift */
           DR(1)
           DR(0))
       {
        sim { AC(0 % 6) := AC(1 % 7);
              AC(7)    := AC(0);
            }
       }
end.
```

The above example has been translated into P-graph and its corresponding control synthesis process is analyzed in [Liu 1989].

3.2 THE GLOBAL ARCHITECTURE OF THE DIGITAL SYSTEM IN THE DIADES SYSTEM

For the conventional language compiler system the compilation process can be accomplished by understanding the exact syntax of both the source and target languages. The first thing to do in designing an SLA is to define the global architecture of the target digital circuits. Before addressing any issues of the analysis of the ADL language and its IR, the global architecture of the target digital circuit in the DIADES system and the formula that describes the architecture will be given.

[Definition3.1] A DIADES Digital System (DDS) is one of the digital systems that are synthesized by DIADES and executes the algorithm described by the respective ADL program.

The purpose and the implementation of a DDS can be various, but the global architecture abstraction of DDS has been presumed (see Figure 5 for the global architecture of the DDS.) The task of a DDS is carried out by two main units: DP and CU. Most current high-level synthesis systems use this model [Goos 1990; Hafe 1982; McFa 1983]. Only the behavioral aspects of a digital system are described in ADL. The first step of the DIADES is to analyze ADL program and generate the P-graph notation. P-graph is an intermediate notation that describes both DP and CU in a list format. The DP and CU specifications are generated based on the P-graph [Liu 1989; Yang 1989; Smit 1988].

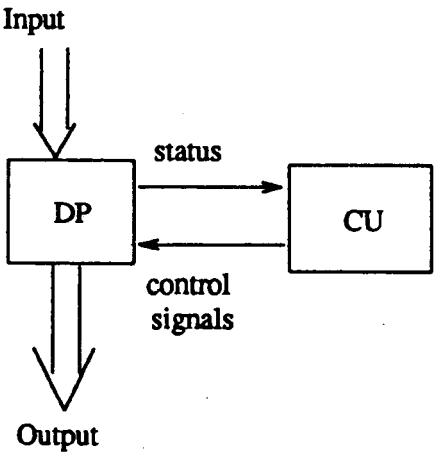


Figure 5. General architecture of the DDS.

Before the P-graph is described, one should know what kind of intermediate notation is needed at this level to describe an ADL program. An ADL program consists of statements with each statement containing two facets: node of the *control flow* and node of the *data flow*. The control flow is another format of the control unit, while the data flow

is another format of the data path. Both the control flow and the data flow consist of *nodes* and *arcs* [Liu 1989]. For both the control flow and data flow an abstract formula is needed to represent a node and its arc that connects the node and its *immediate successor* [Liu 1989]. The control flow node and its arc are represented in this thesis by the *micro instruction (MI);* the data flow node and its arc are represented by the *micro operation (MO).* The list of the micro instructions represents the control flow and the list of the micro operations represents the data flow.

[Definition3.2] An MI is defined as a 6-tuple: *<ID, TY, NE, BR, S, TI>*

where

- ID is the label of the MI;
- TY is the addressing type of the MI;
- NE is the label of the successor MI ;
- BR is used only for a conditional type MI and represents the label of the branch MI;
- S is the set of the MOs signaled by this MI;
- TI is the execution time of the MI.

In the above definition, *TY* can be one of:

- seq -- *sequential* type
- con -- *conditional* type
- jum -- *jump* type
- for -- *fork* type
- joi -- *joint* type (corresponding to *fork* type)

S is a set that could be *NIL*, of one element, or of more than one element; *TI* is an integer that represents the DIADES time unit. *TI* is not currently used because it is not reflected in P-graph.

[Definition3.3] An MO is defined as a 5-tuple: *<ID, I, O, OP, S>*

where

- ID is the label of the MO;
- I is the set of source elements;
- O is the set of destination elements.
- OP is the operator mnemonic;
- S is the label of the MI that controls this MO;

In ADL there are four kinds of operators for OP:

- Arithmetic Operators: +, -, *, / .
- Logical Operators: AND, EXOR, NAND, OR, NOR, XOR, XNOR, and NOT.
- Relational Operators: >, <, =, NE, >=, <=.
- *ASL* (ADL System Library) built-in blocks: Math functions, and some other complex operation blocks.

The above MO definition mentions the source and destination elements. An element is a storage unit. A storage element in TAG90 is represented by VAR (variable), which is a 4-tuple: *<MN, ST, SC, BL>*

where

- MN is the mnemonic of a storage element.
- ST is the type of the storage element, and can be either *int* (integer), *logic* (logical), or *user-defined* type.
- SC is the scope of the storage element, and can be either *input*, *intern*, or *output*.
- BL is the bit length of the storage element.

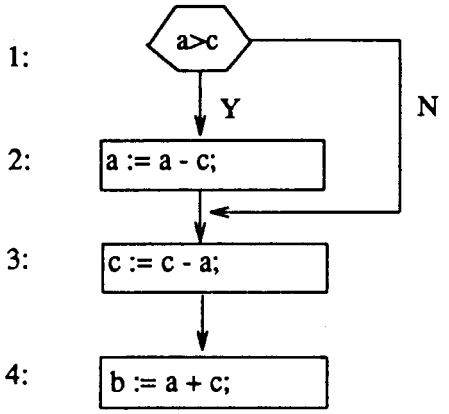
Each ADL statement contains an MI and an MO which are machine (target code) independent, and suitable for conceptual intermediate representation of CU and DP. For

example, several ADL statements and the corresponding data flow graph are shown:

The ADL Statements:

```
1:   if (a > c)
2:   a := a - c;
3:   c := c - a;
4:   b := a + c;
```

The Data Flow Graph:



The above ADL statements can be decomposed into 4 MIs :

```
(1 con  2 3 (0) -)      /* MI1: MI for if (a > c)     */
(2 jum  4 0 (1) -)      /* MI2: MI for a := a - c;    */
(3 seq  4 0 (2) -)      /* MI3: MI for c := c - a;    */
(4 seq  5 0 (3) -)      /* MI4: MI for b := a + c;    */
```

and 4 MOs:

```
(0 (a c) (ALU_com) > 1)  /* MO0: MO controlled by MI1 */
(1 (a c) (a) - 2)        /* MO1: MO controlled by MI2 */
(2 (c a) (c) - 3)        /* MO2: MO controlled by MI3 */
(3 (a c) (b) + 4)        /* MO3: MO controlled by MI4 */
```

In the above example, it is noteworthy that in the MO, the sequence of the elements in the resource list is significant. For operators such as "-" and ">", different sequences have different operational semantics. In MO number 0, *ALU_com* is assumed to be the output of the ">" comparison operation.

3.3 P-GRAPH NOTATION

The P-graph notation is the IR at the first stage of the DIADES silicon compilation process. The reasons for an IR such as the P-graph are:

- The IR of a HBDL provides more details to refine the stepwise top-down design

process.

- Different transformations and optimizations are needed at this level.
- The IR realizes the retargetability of the target digital system.

As mentioned above, the P-graph is also based on the *MI-MO-VAR* model. For historical reasons the P-graph notation is not exactly the same as MI-MO-VAR; however, it is equivalent in representing the DDS architecture. The P-graph notation contains three main lists and other miscellaneous lists which describe the data flow, control flow, memory units, and other features of the DDS.

3.3.1 Control Flow List *coplisset*

coplisset is similar to the list of MIs. The elements in *coplisset* have a 3-tuple format: *<TY, ID, NE>*. TY, ID, NE are identical to that in the MI definition. TY can have three values:

- × -- the same as *seq* and *jum* of TY in an MI. For these two kinds of addressing type, explicit addressing is employed. Therefore, there is no need for TY to distinguish between *seq* (sequential) and *jum* (jump). For example, the element *(× 3 13)* means that *node3* has a successive node *node13*.

- an integer -- the same as *con* (conditional) of TY in an MI. This integer represents the number of a predicate. For example, *(3 4 7)* indicates that node 4 is a predicate node and if predicate 3 is fulfilled, the next node will be node 7. For any predicate node there exists a node that indicates the failure of the predicate. TY of this kind of node is *(not n)* where *n* is a predicate number. For example, *((not 3) 4 9)* corresponds to node 4 in the case that predicate 3 fails in node 4. This means a branch from node 4 to node 9.

- e -- the type symbol for the *fork statement.*

From the above analysis of TY of *coplisset* one can understand why the field *BR* is not necessary in *coplisset*. Field *TI* of MI is not implemented in DIADES yet. It is assumed that all DDS MOs take the same time to be executed, an assumption that is not practical and needs to be changed in future versions of the system. Field *S* of MI is not attached to the *coplisset* element. It is the element of list *nalisset*.

3.3.2 Data Flow List *nalisset*

nalisset is similar to the list of MOs. An element of *nalisset* has the format:

(S (:= O (EXPR)) ..... (1)
(S (S1 S2)) ..... (2)

where in format (1)

- S is the node of *coplisset* that controls this operation.
- := is the assignment operator, which indicates the destination of the operation.
- O is the destination element of the operation.
- EXPR is the ADL arithmetic expression.

For example, *(5 (:= y a))* means that node 5 of *coplisset* specifies the register transfer operation $y := a$. Some different node of *coplisset* may have the same operation; however, format (2) deals with this situation. For example *(7 (5 6))* means the operations specified by both node 5 and node 6 are contained in node 7.

3.3.3 Memory Unit List *lzmset*

lzmset list is the list of all memory units used in the DDS. An element of the *lzmset* has the format: $(v_1\ v_2\ \ldots\ v_n)$ where v_i represents the mnemonic of a variable of the ADL program.

3.3.4 Other Lists

Other lists of the P-graph include:

1. A list of predicates (contained in *plisset*). Each element in this list represents a predicate number as well as the predicate specified by it. For example, *(10 (lessp x 20))* means that predicate 10 implies the relation "$x < 20$."

2. A list of description of node properties (contained in *nolisset*). The property of each node is described in this list and each element has the format as either *(cond number nill)* or *(number number nil)*. For example, *(cond 8 nil)* means that node 8 is a predicate node; *(5 5 nil)* and *(5 12 nil)* indicates that both node 5 and node 12 are operational nodes and specify the same operations as those specified in node 5.

The P-graph notation is quite different from the MI-MO-VAR notation system described above; however, it is reasonable for a P-graph to continue to exist because most DIADES programs at lower levels are based on P-graph.

CHAPTER IV

APPLYING OBJECT-ORIENTED PROGRAMMING TO HIGH-LEVEL SYNTHESIS

This chapter focuses on the object-oriented design methodology employed in the PCG design. A new model of SLA design is established in this thesis, which involves a new set of concepts and abstract formulas. The conclusion of this chapter is that this model has improved the software design methodology of high-level synthesis system design and has several advantages over the traditional top-down design approach.

4.1 CONCEPTS OF OBJECT-ORIENTED DESIGN AND PROGRAMMING

Object-oriented design focuses on the data to be manipulated rather than on the procedures that do the manipulation. Object-oriented programming languages support the following concepts [Wien 1988]:

- Data Encapsulation: No direct access to data representation (also known as data hiding.)
- Message Passing: Access to and manipulation of data are accomplished by passing messages to the encapsulated entity so as to invoke the appropriate actions.
- Inheritance: Creation of new forms of data and code combinations in a systematic way from previous combinations.
- Polymorphism: The ability to refer at run-time to instances of various classes, i.e., the same function name may bind to a different code when applied to different objects at run-time.

In the next section, an abstract object-oriented model is established to describe the PCG design methodologies. This abstract model is not based on a particular OOP language such as *Smalltalk, Eiffel,* or C++, it is based on the general concepts supported by most object-oriented languages.

4.2 OBJECT AND ITS FORMAL MODELS

The abstract model of objects and relations between objects will be presented in this section. The definitions of object, attribute, method, and class are from [Kim 1990]. The concepts of procedure and function are from [Meye 1988].

4.2.1 Core of the Object-Oriented Data Models

> *Object and Object Identifier:* In object-oriented systems and languages, any real-world entity is uniformly modeled as an *object*. Furthermore, an object is associated with a unique identifier [Kim 1990].

The object identifier is used to pinpoint an object to retrieve.

> *Attributes and Methods:* Every object has a state and a behavior. The *state of an object* is the set of values for the attributes of the object, and the *behavior of an object* is the set of methods (program code) which operate on the state of the object.
> *Class:* A class is specified as a means of grouping all the objects which share the same set of attributes and methods. An object must belong to only one class as an instance of that class. The relationship between an object and its class is the *instance_of* relationship. A class is similar to an abstract data type [Kim 1990].

An instance of class *C* can be called a *C-object*. The notion *C-object* will be used throughout the rest of this thesis to denote an instance of class *C*. For an object *o*, the notation *Class(o)* denotes the corresponding class of *o*.

> *Methods:* Methods are implementations of operations on the instances of a class. There are two kinds of methods:
> • A *procedure* performs an action, which means it may change the state of an object.
> • A *function* computes some value deduced from the state of the object. The notion of state here is simple: the values of the data at any point during system execution determine the state of the object. A procedure call may change this

state, namely the values of one or more fields; a function call returns a value computed from the state (namely from the value of the data) [Meye 1988].

A special method called a *constructor* is a procedure that constructs a class object from the data it needs. A constructor generates an object from the input data needed for forming the initial state of an object. In *C++* code a constructor of class *X* is a procedure with a name as the class name, namely *X*. The definition of the constructor of class *X* is:

X(p) { .... }

where *p* is the list of parameters needed for constructing an *X-object*.

4.2.2 ADL Object and its Formal Models

As mentioned in Chapter I, most current object-oriented data models in the high-level synthesis area are at the RT level. [Lips 1986] describes the *design entity*, the principal hardware abstraction in VHDL which provides for effective separation of interface and function, thus allowing hierarchical design decomposition. In this thesis, the author claims that the object-oriented data model is the best candidate to implement the design entity. Also included in this section is *ADL Object* as a data model for a DDS design entity.

4.2.2.A The Concept of an ADL Object. Firstly, the definition of an ADL object is given:

[Definition4.1] An ADL Object (AO) is a design entity of which a DDS is composed.

The AO is an abstraction of a collection of logically related aggregates of data, without regard to their internal structure. There are several relationships between AOs. Some operations are also defined to transform the states of AOs. For the sake of convenience *object* refers to *ADL Object* for the remainder of this chapter.

An object is a three-tuple: $<I, C, S>$

where

- I denotes the identifier of the object, which is a unique symbol for all objects.
- C denotes the class type to which the object belongs.
- S denotes the state of the object at this moment.

The set of states of an object o (an object whose identifier is o) is denoted as $S(o)$, where $S(o) = \{s_i | s_i$ is a state of $o\}$. It is also consistent to denote the set of states of object o as $S(A)$, if o is_instance_of A. Actually, the notion $S(A)$ is more universal.

4.2.2.B The Relationship between Object Classes. There are three distinct relationships illustrated below.

Inherit_from: The relationship *inherit_from* is denoted as $A \subset B$, where B is a super class of A or A is derived from B. In the inheritance relationship a class that inherits (directly or indirectly) from class C is said to be a *descendant* of C and C is said to be the *ancestor* of its *descendants*. A class is considered to be one of its own descendants. The inherit_from relationship is the most common relationship in OOP. In C++, for example, the inherit_from relationship is realized by class derivation. For example, $B \subset A$ is represented in C++ by:

```
class B: public A
 {
  <class_attributes_and_methods>
 }
```

[Definition4.2] The notation Family(C) represents the *family of class C*, where Family(C)={A| A ⊂ C}.

Composed_of: An object is either primitive or composite. Primitive objects cannot be further decomposed into other objects. On the other hand, composite objects are formed from primitive and other composite objects. The relationship *composed_of* is denoted by $A \ll B$, where A is an attribute of B. In OOP languages the composed_of relationship is realized by declaring other classes as its attributes. In C++, for example,

$A <<C$ and $B <<C$ can be represented as:

```
class C
 {
  A member1;     // A is a class
  B member2;     // B is a class
  <other_part>
 }
```

Supervised_by: The relationship *supervised_by* is denoted by $B < A$, where class A is called the *supervisor* of class B and B is said to be supervised by A. In this relationship the objects of the supervisor of a class are able to access the attribute data of the instances of the class. In C++, for example, the *supervised_by* relationship is realized by a *friend* declaration. A class can choose another class to be its supervisor by declaring the chosen class to be its friend. For example, $A <B$ is represented in C++ by:

```
class A
{
 friend B;
 <other_attributes_and_methods>
}
```

Figure 6 shows the pictorial notation of the above three relationships between objects. The upward arrow represents the relationship inherit_from, the downward arrow represents the relationship composed_of, and the dotted upward arrow represents the relationship supervised_by. The pointing direction of arrows in Figure 6 tells which side of the relationship has the right to choose this kind of relationship. For example, in a $B < A$ relationship, only B has the right to choose A as its *supervisor*, while A has no right to choose to supervise B.

Examples of the above relationships. The inherit_from relationship captures the generalization relationship between a class and its direct and indirect subclasses. For example, a Finite State Machine (FSM) and a Microprogrammed Controller (MC) are two subclasses of class Control Unit (CU). The CU is the generalization of the FSM and the MC and the FSM and the MC are the specification of the CU. More importantly, the

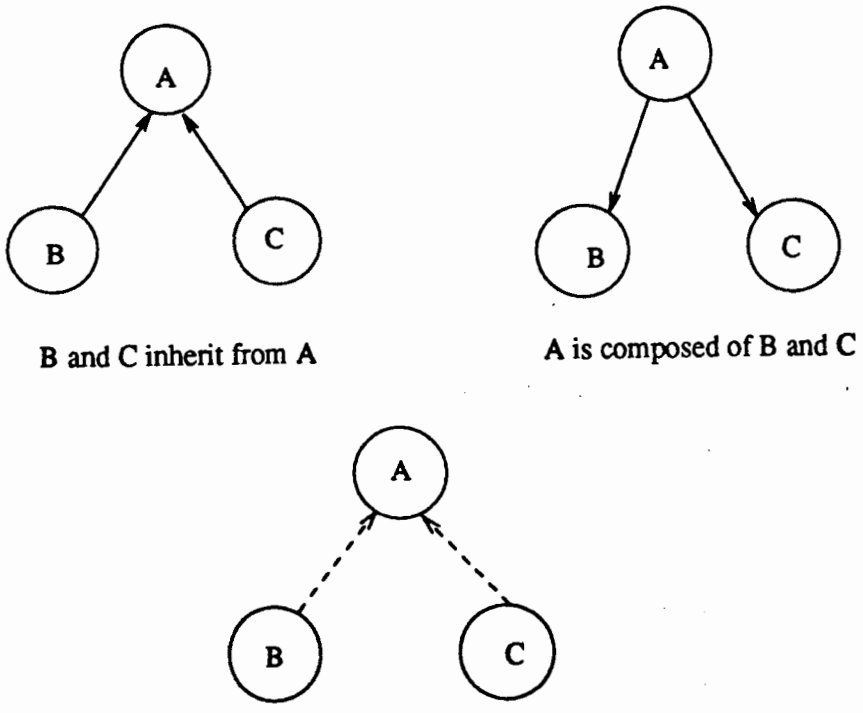


Figure 6. The relationships between classes.

sub-classes can inherit attributes and methods from their superclasses. This class hierarchy is the key to software reusability and extensibility.

The composed_of relationship usually has nothing to do with the inheritance of attributes and methods. This relationship captures the composition relationship between a class and its attributes. For example, a digital circuit is composed of a DP and a CU; therefore, it is said that the digital system has two attributes: a DP and a CU.

The supervised_by relationship is not mentioned by most current literature in OOP. It is, however, an important relationship between ADL classes. The supervised_by relationship seems to be particularly useful in the SLA design. For

example, in TAG90 class DP and class CU are supervised by class DDS. Class DDS has a method to access the attributes in both classes DP and CU, which allows for the interconnection between the CU and the DP and the optimization that is related to both DP and CU. The supervised_by relationship seems to be similar to the composed_of relation, but there are different aspects between these two relationships. The differences are:

- The supervisor class A of class C does not contain a *C-object* as its attribute, however an *A-object* can access the attributes of *C-object*, which can also be realized by the relationship $C<<A$. It is obvious that in software implementation the supervised_by relationship uses less storage space than in the composed_of relationship.

- The supervised_by relationship can also be inherited through the class hierarchy as in a composed_of relationship, but at a much lower price. For example,

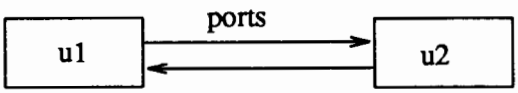
$$\text{if } B < A \text{ and } C \subset A$$

then

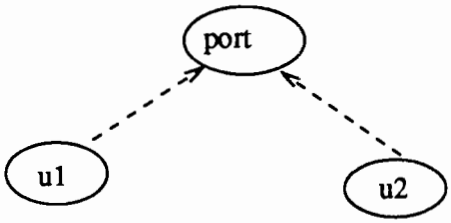
$$B < C.$$

This inheritance of supervised_by relationship can be realized by C inheriting methods that access attributes of B from A.

- The supervised_by relationship extends the horizontal communication among classes. This communication is crucial in modeling hardware design entities, since the objects (namely the design entities) always interconnect with each other in hardware design. In software design this interconnection can be realized by defining a *port* class that supervises the interconnected objects. The pictorial illustration of the *port* class is shown in Figure 7.



u1 interconnects with u2



Objects u1 and u2 are supervised by object port

Figure 7. The supervised_by relationship in hardware design.

4.2.2.C The Transformations on the Objects. Three types of transformations on objects will be introduced: *object modification*, *object querying* and *object synthesis*. There are two ways to denote a transformation: one way is the *code notation* and the other is the *semantic notation*. The code notation reflects the programming aspects of a transformation, while the semantic notation interprets the code notation by giving a semantic interpretation to the transformation. There are three types of transformations on the object:

Object Modification: As mentioned above, a procedure is an action that changes the state of an object. The *prototype* of a procedure consists of the *name* of the procedure, the *parameters* of the procedure and the *class* to which the procedure belongs. In this thesis, a procedure of class C with name f is denoted as

$$\mathbf{P}_C^f$$

It is assumed that different procedures in the same class cannot use the same name; therefore, the above notation represents a distinct procedure.

The definition of procedure $\mathbf{P}_C^f$ is as follows:

$$\mathbf{P}_C^f : PL \times S(C) \rightarrow S(C)$$

where $\mathbf{P}$ means procedure, C is the class type of the procedure, f is the procedure name, PL is the set of all possible parameter values and $S(C)$ is the set of all states of *C-object.*

The code notation of applying a procedure to a *C-object* is:

$$f(p)\_o$$

where f is a procedure, p is a parameter array, and o is the object f is applied to. The semantic notation of the above code notation replaces f by $\mathbf{P}_C^f$, o by $<o\ C\ n>$; therefore, the semantic notation of code notation $f(p)\_o$ is:

$$\mathbf{P}_C^f\ (p)\_<o\ C\ n> \quad \text{..... (1)}$$

If the result of (1) needs to be illustrated, (1) can be extended as:

$$\mathbf{P}_C^f\ (p)\_<o\ C\ n> = <o\ C\ n'> \quad \text{..... (2)}$$

"=" means "the result is." In (2) the result is $<o\ C\ n'>$, where n' is a state of the object o, and $n' = f(p,n)$. Formula (2) illustrates a state change $<n, n'>$ within *C-object* o. For example, assume H is a hash table class; k is a procedure of H, which inserts an element into the hash table; an instance of the hash table is h and v is a value to be inserted to the hash table. The code notation of applying k to h is:

$$k(v)\_h$$

The semantic notation for the above code notation is:

$$\mathbf{P}_H^k\ (v)\_<h\ H\ n1> = <h\ H\ n2>$$

The above semantic notation illustrates that applying procedure k to a hash table h changes the state of h from $n1$ to $n2$, because a new element v is added into h.

Object Querying: As mentioned above, a function computes some value deduced from the state of the object. The prototype of function also consists of three factors: the *name* of the function, the *parameter* of the function and the *class* to which the function

belongs. A function g of class C is denoted as

$$\mathbf{F}_C^g$$

The definition of function $\mathbf{F}_C^g$ is as follows:

$$\mathbf{F}_C^g : PL \times S(C) \rightarrow V$$

where $\mathbf{F}$ denotes function, C is the class type of the function, g is the name of the function, PL is the set of all possible parameter values of g, $S(C)$ is the set of all states of *C-object*, and V is the set of all possible values deduced from the state of *C-object*. The code notation of applying a function f to a *C-object* o is denoted:

$$g(p)\_o$$

where g is a function, p is a parameter array, o is the *identifier* of the object that g is applied to. The semantic notation of the above code notation replaces g by $\mathbf{F}_C^g$, o by $<o\ C\ n>$, therefore, the semantic notation of the code notation $f(p)\_o$ is:

$$\mathbf{F}_C^g\ (p)\ \_<o\ C\ n> \quad \text{..... (3)}$$

If the result of (3) needs to be illustrated, (3) can be extended as:

$$\mathbf{F}_C^g\ (p)\ \_<o\ C\ n> = v \quad \text{..... (4)}$$

"=" means "the result is." In (4) the result is v, where $v = g(n)$, n is a state of the object o; therefore, (4) shows a computation $<n, v>$ from a *C-object* at state n.

Again, take the above hash table as an example and assume that *pr* is a function of class H that returns a list of all elements stored in the hash table. The code notation for applying function *pr* to a *H-object h* is:

pr()_h

The semantic notation for the above code notation is:

$$\mathbf{F}_H^{pr}\ ()\_<h\ H\ n> = <list\_of\_elements>$$

Object Synthesis: This transformation shows how an object is constructed from other objects. A transformation $\mathbf{S}_C^+$ is defined to denote the operation of *synthesis*, which

constructs a *C-object*. The code notation for the synthesis transformation is:

$$\mathbf{S}_C^+ (O_1, O_2, ..., O_i) = O$$

where $Class(O) = C$, $Class(O_n) \ll C$, and $1 \le n \le i$. The semantic notation for the above code notation is:

$$\mathbf{S}_C^+ (<o_1\ C_1\ n_1>, <o_2\ C_2\ n_2>, ..., <o_i\ C_i\ n_i>) = <O\ C\ n>$$

where $C_k \ll C$ and $1 \le k \le i$. This operation allows the synthesis of new objects from the existing objects. The synthesis transformation is the construction of an object; therefore, it is called the *construction transformation.*

Two rules for pure transformations: Object modification and object querying are said to be "pure" transformations. A pure transformation is denoted as $\mathbf{T}_C^f$.

A transformation's code notation and its semantic notation can be connected by $\equiv$ in a format $cn \equiv sn$, which means the semantic notation sn interprets the meaning of the code notation cn. For example,

$$f()\_o \equiv \mathbf{T}_C^f\ ()\_<o\ C\ n>$$

means that the semantic notation for transformation $f()\_o$ is $\mathbf{T}_C^f\ ()\_<o\ C\ n>$.

There are two rules for $\mathbf{T}$:

1. Polymorphism Rule: Polymorphism means that a single format has different meanings.

[Definition4.3] An object variable (a variable for short) is an alias for an object. A variable can be bound to an object. In this case the variable is the alias of the bound object and the object is said to be the value of the variable.

There are three aspects to a variable: its name, its class type, and its value. To declare a variable is to assign a class type to the variable. For example, declaring a variable x to be class C type is denoted in C++ language as:

$$C\ x;$$

The function *D_class(x)* returns the declared class type of variable x.

Assigning a value to a variable means binding this variable to an object. The assignment is denoted by operator ":=". For example, to assign an object <i C n> to variable x is denoted as:

$$x := i \text{ or } x := <i\ C\ n>$$

(1) Polymorphic assignment rule:

if $D\_class(x) = C$, then $x := <i\ B\ n>$ is a legal assignment *iff* $B \subset C$.

This rule allows a variable to be assigned any objects that are instances of a class family as its value. The class family is determined by the previously declared class type of this variable. When a variable is assigned an object, the class type of this object becomes the actual type of the variable; hence, the class type of variable x is not necessarily of one type, but can be a set of types. The set of the class types of variable x is denoted by TYPE:

$$TYPE = Family(C) \text{ where } C = D\_class(x)$$

(2) Polymorphic transformation rule:

If x is a variable, $D\_class(x) = C$, and f is either a procedure or a function, then

$f(p)\_x \equiv \mathbf{T}_B^f\ (p)\_<i\ B\ n>$,

iff:

(1) f is a *virtual* method declared in C,

(2) B is_in Family(C), and

(3) f is also declared in class B.

In the above formula the transformation $f(p)\_x$ is dynamically interpreted according to the different bindings of x. The formula shows a powerful mechanism of program extensibility. For example, a function *getId* is designed to return the class name of any object. The code notation for applying this function to an object variable *Obj* is:

getId()_Obj

If:

(1) *getId* is defined as a *virtual* function in a super class called *ROOT*,

(2) D_class(Obj) = ROOT,

(3) $SUB \subset ROOT$,

(4) and *getId* is also defined in *SUB* as getId() = "SUB" ("SUB" is the name of class SUB.)

then

$$getId()\_Obj \equiv \mathbf{F}_{SUB}^{getId}\ ()\_<i\ SUB\ n> = \text{"SUB"}.$$

This example shows that the same code can be interpreted differently.

2. Supervision Rule:

The supervision rule for object transformations is as follows:

If $C < A$ and a *C-object* $<o\ C\ n>$ is a parameter of transformation $\mathbf{T}_A^f$ of class A, which has the semantic notation of

$$\mathbf{T}_A^f\ (<o\ C\ n>)\_<o'\ A\ n'> \quad \text{..... (5)}$$

then there exists a method of C, $\mathbf{T}_C^g$, which is applied to object $<o\ C\ n>$ and produces the same result as (a). This will be denoted as:

$$\mathbf{T}_C^g\ \_<o\ C\ n> = \mathbf{T}_A^f\ (<o\ C\ n>)\_<o'\ A\ n'> \quad \text{..... (6)}$$

The formula (5) means that the supervisor class A of C has a method f to access the attributes of class C. The formula (6) means that method f is equivalent to the method g of class C. For example, for $A < B$ there is a function *getbV* of A that takes a *B-object* b as its argument and prints all the attribute values of the *B-object*. Function *getbV* is a method of A, thus must be applied to an *A-object* $a = <o\ A\ n>$, even if it will have no side effect on this object. The process for a to get to print the attributes of b is the following:

getbV(b)_a

It is obvious that there is a method of B, *printB*, which will cause the same effect as *getbV* does; therefore,

$$getbV(b)\_a = printB()\_b.$$

This example shows that the supervision class can have accessibility to the data of the classes supervised by it and achieve the same functionality as the class methods, which would have been especially created.

4.3 THE ADL CLASS STRUCTURE

The abstract model of ADL Objects was introduced in the last section. The ADL classes and the relationships between them will be presented in this section.

4.3.1 The Structure of ADL Classes

The central concept of OOP is the analysis and implementation of the data models underlying the system to be designed. In this chapter the design entity within the DDS is modeled as an ADL Object. The structure and organization of the classes of the ADL Objects will be presented in this section.

The Root of the System: The root of the class structure is the ancestor of every ADL class. The root of ADL classes is denoted by ROOT. The relation between ROOT and other classes is

$$C \subset ROOT \text{ whenever } C \text{ is an ADL class.}$$

Therefore, any class in this structure can be viewed as a member of Family(ROOT).

The Architecture Aspect: Class SYSTEM and its Descendants: ADL is an algorithm-oriented, high-level hardware description language. The *microarchitecture* (i.e., the structure consisting of registers, buses, RAMs and ALUs) is not explicitly specified by the behavioral description; however, the global architecture of the DDS is

implicitly predefined. This global architecture needs to be represented in the class structure. Class SYSTEM is an abstract class representing the architecture of the target digital system. Under class SYSTEM are the abstract class DDS and the two main design entity classes, CU and DP.

DDS is an abstract class reflecting the architecture of the target digital system in the DIADES system. Classes DP and CU are supervised by class DDS, they represent two main design entities of the DIADES system: the data path and the control unit. The relationship between these three classes under class SYSTEM is

$$CU < DDS \ AND \ DP < DDS.$$

Classes DP and CU contain composite and primitive design entities, the latter belonging to classes under the class PRIMITIVE. Class DDS contains the global information of the DDS and has access to the data and methods in both CU and DP.

The Functional Aspect: Class PRIMITIVE and its Descendants: There are various design styles and design methodologies for functional elements design using a high-level synthesis system. In this thesis the formula MI-MO-VAR represents the design entities of the functional level architecture.

Class PRIMITIVE is an abstract class dealing with design entities at a lower level compared with the classes of the SYSTEM family. Formula MI-MO-VAR defined in Chapter III corresponds to three main design entities at this level. Class MI represents the microarchitecture of the control unit. In the DIADES system MI is in the form of microinstructions which is similar to that of a microprogrammed control unit. An MI can also be viewed as a node in the control flow. Class MO represents the data computation and the communication path between data resources. An MO can be viewed as a node in the data flow. Class VAR represents the abstract storage element defined in the ADL program. Class LIB_ELEM is a *pseudo class* type, which represents the element in the ADL System Library (ASL). LIB_ELEM is not a member of Family(PRIMITIVE);

however, it has supervised_by relationship with class PRIMITIVE. The ASL elements will be discussed in Chapter V in greater detail. The relationship between classes under class PRIMITIVE and class LIB_ELEM is

$$LIB\_ELEM < PRIMITIVE .$$

Class VAR, MI and MO objects are able to supervise lower level information stored in the ASL through the supervised_by relationship between classes LIB_ELEM and PRIMITIVE. This allows the transformations and optimizations at the functional level to take into account the bottom information at an early stage. For the same reason class LIB_ELEM is also supervised_by class SYSTEM.

The Directed Acyclic Graph (DAG) Structure of Classes. Figure 8 shows the DAG diagram of ADL classes.

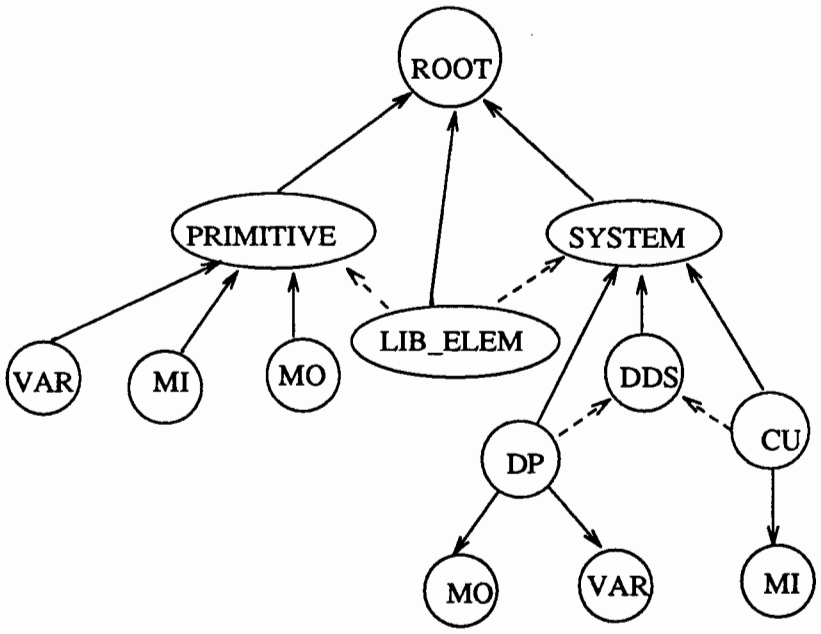


Figure 8. The ADL class DAG.

In Figure 8 all classes described in the above section are shown. The relationships between them are visually presented. Usually, the inherit_from relationship is the

only relationship between classes in an object-oriented programming language [Mili 1990]. As a new contribution to the SLA design, the DAG of ADL classes and the relationships composed_of and supervised_by are introduced. This class structure model can improve the expressionability of an object-oriented system that realizes a high-level synthesis system.

4.3.2 ADL Classes Close-up

ADL classes are the abstract types of the ADL objects. For the remainder of this chapter an ADL class will simply be called class for the sake of convenience.

The abstract diagram of a class is :

{class:

class_name

inherit_from:

parent_name

supervised_by:

supervising_class

attributes:

(a1, a2, ..., an)

methods:

(p1, p2, ..., pi)

(f1, f2, ..., fj)

}

In the above diagram, p_i denotes a procedure; f_i denotes a function. In this thesis, the following classes are the recognized types of the design entity of the DIADES system (the remarks following ";" are the comments):

- class VAR represents the storage unit of a DDS. The diagram of VAR class definition is:

```
{
  class:
        VAR
  inherit_from:
        PRIMITIVE
                        ;PRIMITIVE is the parent class of several classes
  attributes:
        (
                        ; list of attributes
        var_name,
                        ; name of a VAR-object
        var_type,
                        ; type of a VAR-object
        var_scope,
                        ;scope of a VAR-object, one of
                        ;input, output, or internal
        var_value,
                        ;value of the variable held in the storage unit,
                        ;usually not used
        var_length.
                        ;the bit length of the storage unit
        )
  methods:
        (
                        ;list of procedures
        VAR_constructor,
                        ;VAR constructor
        set_attribute
                        ;sets the values of attributes
        )

        (
                        ;list of functions
        print
                        ;prints out the VAR-object in the given format
        )
}
```

- class MO represents a primitive data operation (i.e., the node of the data flow.)

```
{
  class:
        MO
  inherit_from:
        PRIMITIVE
                        ;Class MO is a member of Family(PRIMITIVE)
  attributes:
        (
                        ; list of attributes
        id,
                        ; the label of the MO
        op,
```

```
                    ;arithmetic operator such as +. -, *, etc.
        cs,
                    ;control signal that evokes this MO
        i_set,
                    ;input list, set of class VAR objects
        o_set
                    ;output list, set of class VAR objects
        )
    methods:
        (
                    ;list of procedures
        MO_constructor,
                    ;the constructor of MO
        set_value
                    ;sets the attributes for the MO-object
        )

        (
                    ;list of functions
        print
                    ;the function that prints the MO-object in the given format
        )
}
```

• class MI represents the node of the control flow of a DDS. The diagram of MI class definition is:

```
{
  class:
        MI
  inherit_from:
        PRIMITIVE
                    ;inherits features from PRIMITIVE
  supervised_by:
        SYSTEM
                    ;supervised by its superclass
  attributes:
        (
         id,
                    ;the label of the MI
        ty,
                    ;the addressing type of the MI
        ne,
                    ;the label of the successive MI
        br,
                    ;the label of the branch MI (used in conditional MI)
        mo_list,
                    ;the set of MOs signaled by this MI
        ti
                    ;the execution time of the MI
        )
```

```
    methods:
        (
                        ;list of procedures
        MI_constructor,
                        ;the constructor of MI-object
        set_attributes
                        ;set the attributes of MI-object
        )


        (
                        ;list of functions
        print()
                        ;the method that prints a MI-object in a given for-
                        mat
        )
}
```

• class LIB_ELEM is a pseudo class of the ASL elements. The diagram of class LIB_ELEM definition is:

```
{
  class:
        LIB_ELEM
  inherit_from:
        ROOT
  supervised_by:
        SYSTEM, PRIMITIVE
  attributes:
        (a1, a2, ..., ai)
                        ;This will be addressed in Chapter V
  methods:
        (
                        ;procedure
                        ;undefined temporarily
        )

        (
                        ;function
        getId
                        ;a function that returns the class type
                        ;of the object which the function is applied to
        )
}
```

• class DP is a composite class. It contains a list of MOs and a hash table of VARs. The diagram of the class definition of DP is:

```
{
  class:
        DP
  inherit_from:
        SYSTEM
                        ;SYSTEM family member
  supervised_by:
        DDS
  attributes:
        (
                        ;The list of attributes
        var_table,
                        ;The hash table that stores VAR-objects
        mo_list
                        ;The list that stores MO objects
        )
  methods:
        (
                        ;The list of procedures
        DP_constructor,
        compress
                        ;The method that executes the algorithm of
                        ;compressing MO list.
        )

        (
                        ;The list of functions
        print_nalisset,
                        ;Form the *nalisset* list of the P-graph
        print_lzmset,
                        ;Form the *lzmset* list of the P-graph
        getId
                        ;return the class type of the object
        )
}
```

A DP is composed of a list of MOs and a hash table of VARs. The procedure *compress* of DP is the major optimization transformation in the class DAG in that it compresses the MO-list according to the data dependency in order to exploit the maximal parallelism of the data flow. This procedure has not yet been implemented. A DP is supervised by DDS, which means a DDS-object has accessibility to the attributes of a DP-object. Function *print_nalisset* uses attributes *mo_list* as input and obtains a list in *nalisset* format. Function *print_lzmset* uses attributes *var_table* as input and obtains a list in *lzmset* format. *Nalisset* and *lzmset* are P-graph lists.

• class CU represents the control unit of a DDS. It contains the list of MIs. The diagram of ADL_CU class definition is:

```
{
  class:
        CU
  inherit_from:
        SYSTEM
  supervised_by:
        DDS
  attributes:
        (
        mi_list,
                        ;List the MI-objects
        )
  method:
        (
                        ;list of procedures
        CU_constructor,
                        ;the constructor of CU-object
        compress
                        ;compresses the MI-list
        )

        (
                        ;list of functions
        print_coplisset,
                        ;creates the *coplisset* list for P-graph
        print_nolisset,
                        ;creates the *nolisset* list for P-graph
        getId
                        ;returns the class type of the object
        )
}
```

A CU is composed of a list of MIs and it is similar to a microprogrammed control unit, which is composed from microinstructions. The procedure *compress* is an optimization transformation on the CU-object. The algorithm for *compress* in this case is similar to the algorithm for the scheduling algorithm used in micoprogramming compaction. CU is supervised by DDS, which means a DDS-object has accessibility to the attributes of CU. Function *print_coplisset* uses *mi_list* as input and obtains a list in **coplisset** format, which is the control flow list for the P-graph. Function *print_nolisset* obtains the *nol-liset* from the *mi_list*.

• class DDS is the supervisor class of class DP and CU. It does not contain a DP-object or a CU-object; however, a DDS-object has accessibility to the attributes of both DP and CU objects. The diagram of class DDS is:

```
{
  class
        DDS
  inherit_from:
        SYSTEM
  attributes:
        (
        name
        )
  methods:
        (
        gen_p_graph
                        ;P-graph code generator
        getId
                        ;return the class type of the object
        )
}
```

DDS is the supervisor for classes DP and CU and is actually an abstract class for a DDS. Since a DDS-object has the right to access all information stored in the DP-object and CU-object, it is able to generate an IR according to the predefined format. Currently, the IR for the DIADES system is the P-graph; therefore, function *gen_p_graph* is defined to generate P-graph. By accessing the methods of DP and CU the lists *nalisset*, *lzmset*, *coplisset*, and *nolisset* of the P-graph are generated. The P-graph list *plisset* needs information in both DP and CU; therefore, in function *gen_p_graph*, the code for generating *plisset* is specified.

• class SYSTEM is the super class of several composite classes and supervises the classes DP, CU, and DDS.

The purpose of this class has two facets:

1. Performs some global optimization;

2. Makes it possible to incorporate new technologies into the DIADES automation system.

The diagram of SYSTEM is:

```
{
  class:
        SYSTEM
  inherit_from:
        ROOT
  attributes:
        (
                              ;list of attributes
        id,
        constraint_list
                              ;list of system constraints
        )
  methods:
        (
                              ;list of procedures
        SYSTEM_constructor
        )

        (
                              ;list of functions
        optimization,
                              ;global optimization
        print,
                              ;prints the data format
        getId
                              ;return the class type of the object
        )
}
```

SYSTEM is the abstract class representing the architectural features of the *target digital system* to be designed. A SYSTEM-object is constructed before any other objects. The attribute *constraint_list* of the SYSTEM is from the compiler macro defined by the user. The function *optimization* is a *virtual* function for global system optimization based on the *constraint_list*. This function is not specified in SYSTEM, but is inherited and implemented by its descendants, such as the DDS.

The DAG structure of classes is the crucial asset of the PCG. This structure offers the following advantages:

1. *Better conceptual modeling.*

Since the conceptual hierarchies are very common in a digital system, direct

modeling of such hierarchies makes the conceptual structure of DDS easier to comprehend.

2. *Factorization.*

   Inheritance supports that common properties of classes are factorized - that is, described only once and re-used when needed. For example, all SYSTEM family members share the properties of class SYSTEM, thus avoiding redundant description.

3. *Polymorphism.*

   The hierarchical organization of the ADL classes provides a basis for the introduction of polymorphism in the sense that the same function name may bind to a different code when applied to different objects at run time. On the other hand, a procedure with a formal parameter of class C will accept any instances of Family(C) as actual parameter.

4. *Stepwise refinement in design and verification.*

   Inheritance hierarchies support a technique where the most general classes containing common properties of different classes are designed and verified first. Then specified classes are developed top-down by adding more details to the existing classes. This feature makes the TAG90 easy to be extended and refined.

4.3.3 The ADL Object Transformations

In the previous section a close-up of all classes was presented. All pure transformations on the objects have been listed within the diagram of each class. Object transformation is the key concept of the PCG design. Although the set of the object transformations is not completed at present because procedures *compress* and *optimize* have not been implemented yet, this set is fully open, namely, extensible. Currently, the set of major pure transformations within the class DAG is as follows:

$S(T) = \{ F^{print}_{VAR}$
F^{print}_{MI}
F^{print}_{MO}
$P^{compress}_{DP}$
$F^{print\_nalisset}_{DP}$
$F^{print\_lxmset}_{DP}$
$P^{compress}_{CU}$
$F^{print\_coplisset}_{CU}$
$F^{print\_nolisset}_{CU}$
$F^{gen\_p\_graph}_{DDS}$
F^{getId}_{SYSTEM} ;virtual function
$F^{optimize}_{SYSTEM}$;virtual function
}

4.4 OBJECT-ORIENTED APPROACH IN HIGH-LEVEL SYNTHESIS SYSTEM DESIGN

As a summary of this chapter, the design methodology used in the PCG design is characterized as a different approach to a high-level synthesis system.

4.4.1 New Design Methodology - Comparing with Top-down Approach

The first step in high-level synthesis is usually the compilation of the formal language into an internal representation. Most approaches use graph-based representations that contain both the data flow and the control flow implied by the specification [McFa 1990-a]. The traditional top-down approach to SLA design is shown in Figure 9. The drawbacks of this approach are:

- *Lack of low level information.* At the translator stage shown in Figure 9, the computational elements are simply pieces of passive data representing fixed black boxes with certain functional capabilities and certain abstract costs. The decisions about what physical modules are to be used and how they are to be placed are deferred until after the RT-level structure has been set [McFa 1990-b].

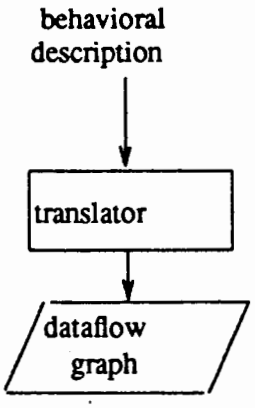


Figure 9. Traditional top-down approach to the SLA design.

- *Lack of extensibility*. The translator is usually procedure oriented and the IR generated by it is procedure and technology dependent; therefore, it is difficult to introduce new technology into the system. A slight change in either the behavioral description format or the IR format usually means a great amount of work inside the translator.

On the contrary the PCG approach of the TAG90 system uses *active code* (ADL objects) to represent the design entities; it has an object-oriented IR, allowing an easy extensibility and maintainability. Due to the employment of the object-oriented data model and programming based on the model, the PCG presents several features different from the traditional top-down approaches. The PCG approach of the TAG90 system is shown in Figure 10.

In Figure 10 the PCG of the ADL analyser is illustrated by the box that contains the *class DAG* and the *object manager*; therefore, the PCG can be viewed as a two-tuple:

$$<L\ M>$$

where L represents the class DAG and M represents the object manager.

The class DAG:

The class DAG consists of the ADL class definition and its corresponding method

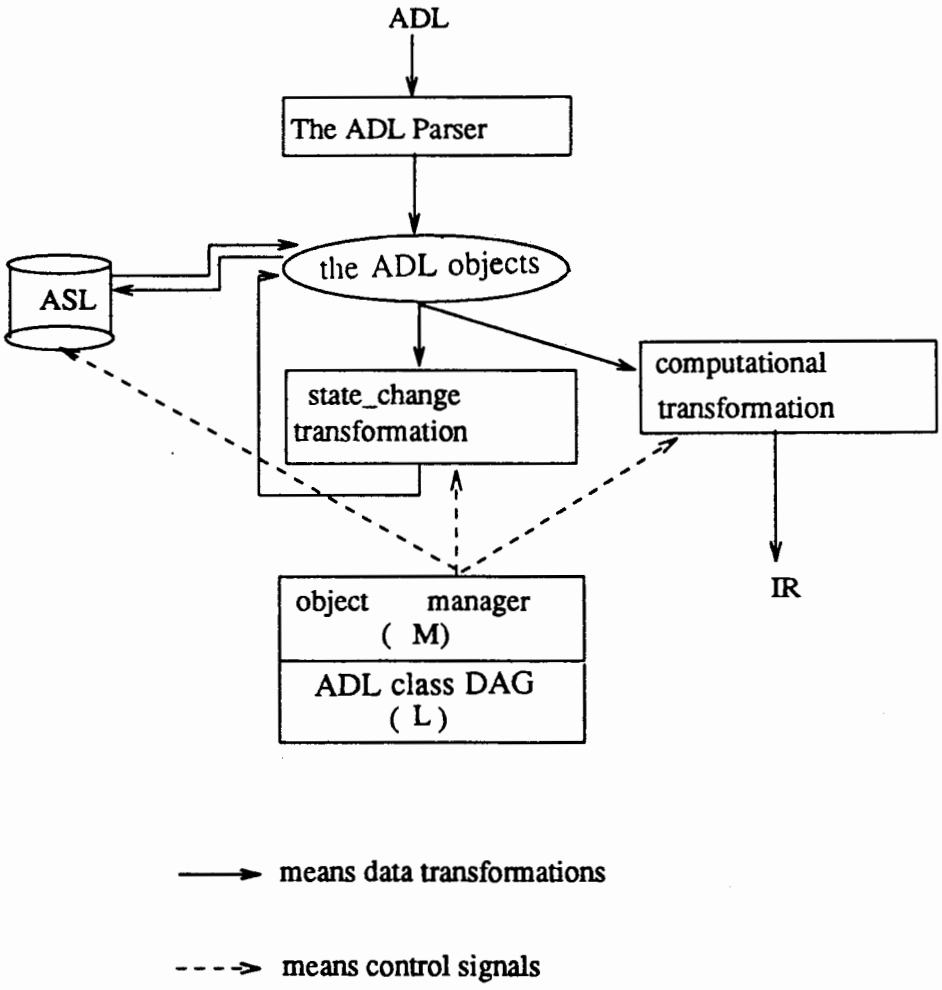


Figure 10. The diagram of ADL analyser.

definition. The design entities of the DIADES system are established by setting up the ADL class DAG. A set of transformations on the ADL objects are designed by defining the methods of the ADL classes.

The object manager:

The object manager consists of semantic routines attached to the YACC grammar rules. The function of the object manager is to manage the process of ADL objects construction and transformation. The construction of the ADL objects is conducted by the class constructor of the class whose instance is to be constructed. The transformations on

the ADL objects are performed by the class methods that are defined in the class DAG. The object manager invokes a class method by sending a message to an object or several objects. For example, the object manager sending a message to the *C-object x* to invoke a transformation called *optimize* can be represented by a piece of *C++* code:

optimize().x

where *optimize* is $\mathbf{F}_C^{optimize}$, D_class(x) = C.

4.4.2 The Advantages of PCG

Using the PCG model, the design tasks are distributed among different object classes. The data and methods are connected and organized through the class DAG. The constructions and transformations of the objects are controlled by the object manager, which is like a post office sending messages to invoke transformations on the elements of the class DAG. The advantages of this new methodology are the following:

a. The object can simulate the design entity at various design levels by encapsulating data and methods and thus becomes an *active design subject* instead of the traditional data model, which is a passive, abstract, and separated *black box* [McFa 1990-b].

b. The design process and the coding process are organized according to the requirements of modern software engineering theory. In fact, the class DAG presented in this chapter is only slightly different from the actual code of the PCG. Similarly, the object manager consists of modular semantic routines attached to standard grammar rules of the YACC specification file. (These techniques will be discussed in Chapter VI.)

c. The greatly improved extensibility is realized. (The extensibility of the TAG90 system will be discussed in detail in the next chapter.)

d. Low level information is accessible at the PCG level. The supervised_by rela-

tionship between objects makes it possible for the objects at the higher level to access the lower level information through the class DAG and the ASL. The supervised_by relationship between the objects seems to be a new contribution to the concept of OOP.

CHAPTER V

THE EXTENSIBILITY OF ADL

5.1 INTRODUCTION

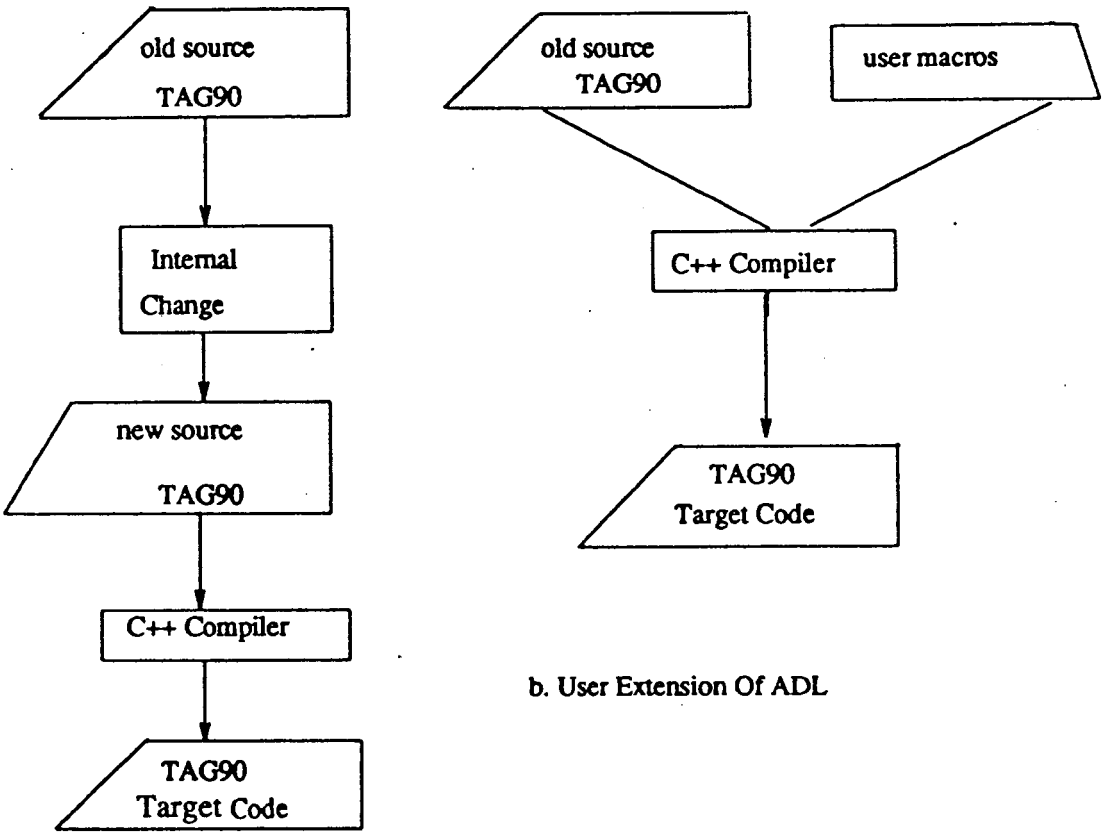
The extensibility of a system is characterized by the following features:

1. The system must be able to cope with a wide variety of application areas. New applications may be created by specializing the existing ones.
2. New solution techniques can be incorporated into the system without modifying the existing models or the knowledge base.
3. The addition of new applications does not depend on the internals of the current system or depends on it only minimally [Kim 1990].

Extensibility is one of the major design goals for a successful HBDL [Aylo 1986]. Generally speaking, any software system has extensibility to some extent. But for a non-object-oriented system, the lack of abstraction complicates adding new applications to the system. Furthermore, the lack of encapsulation impedes the modification of the system itself, because applications depend on the system internals. The OOP design provides a regular and systematic way of system extension. The extensibility of SLA will be discussed in detail.

There are two kinds of extension processes: *system level extension* and *user controlled extension*. System level extension involves modifications, additions, or deletions of the internal data structures and procedures, and is performed by an expert who knows the internal implementation of the current software system. On the other hand, user controlled extension adds new knowledge to the knowledge base of the system without changing the internal implementation of the system. This extension process can be controlled by the user using simple commands. Both kinds of extension process have to

recompile the compiler of the system. Both kinds of ADL extension process are shown in Figure 11.



a. System Extension Of ADL

b. User Extension Of ADL

Figure 11. Two kinds of ADL extension process.

Conventionally, *the extension of a system* refers to the system level extension, as shown in Figure 11a. The user-controlled extension method (Fig. 11b) proposed in this thesis is one in which the user controls and manages the extension of ADL.

5.2 THE SYSTEM EXTENSION OF ADL

5.2.1 Simple Syntax Extension

The TAG90 system is modularized on the basis of ADL grammar rules (see Chapter II); the PCG is modularized on the basis of ADL objects. Therefore, syntax extension is easily performed if the extended part does not require new ADL objects. The addition of new control constructs into ADL is such an extension. Practically, TAG90 was built in this way: in the beginning it could analyse only a subset of ADL, then it was extended step by step to its current state. For example, *for-loop* is introduced to ADL control construct without affecting any other part of the PCG. In the ADL parser specification file, the grammar rule for *for-loop* is inserted with action routines:

```
for_statmt
        : FOR '(' for_initstmt sc
                {
                $<y_label>$ = adl_instrct_id.getIid();
                  // The label of the starting address of the
                  // 'for' loop is stored in the value stack.
                }
        expr_end sc
                {
                gen_cond();
                $<y_label>$ = adl_instrct_id.getIid();
                  // generate a 'cond' MI and store the label
                  // of the address of this MI
                }
        expr_change rp
                {
                int i = adl_instrct_id.getIid();
                  // add the number for the current MI
                mod_next( i - 1, $<y_label>8);
                  // For 'expr_change': goto expr_end
                }
        compound_statement
                {
                int i = adl_instrct_id.getIid();
                gen_instrc(i, $<y_label>11, 0, 3, 0);
                  // goto expr_change
                mod_branch($<y_label>5, adl_instrct_id.getIid();
                  // For 'expr_end': if 'no' exit the loop
                }
        ;
```

```
for_initstmt
        : /* maybe empty */
        | assign_stat
        ;
```

The above code is clearly illustrated by Figure 12. The MI_i is generated in the sequence from MI_1 to MI_4 with the relationships between these MIs denoted by the arrows. Each MI is generated inside the rule for the non-terminals *for_initstmt*, *expr_end*, *expr_change*, or *compound_statement*. The *ne* and *br* field of MI_2 need to be modified after MI_4 is generated. The *ne* fields of MI_3 and MI_4 need also to be modified. These modification processes are shown in the aforementioned grammar rules for the *for-loop*.

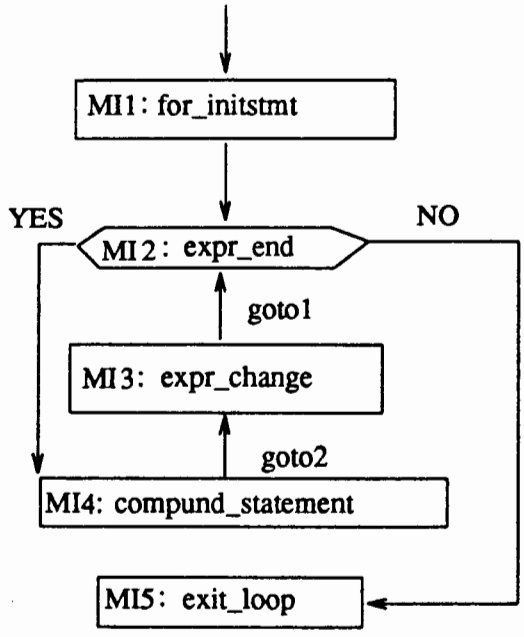


Figure 12. The illustration of the grammar rule of the 'for-loop.'

The insertion of the *for-loop* grammar rules into the Yacc specification file does not affect other parts of the parser to a great extent.

5.2.2 Adding New ADL Objects

The basic techniques for software *reusability* and *extensibility* in OOP design are inheritance and polymorphism. *Reusability* refers to the building of a system on previous accomplishment and extending their results instead of trying to solve every new problem from scratch; it is a special and major way to extend a software system in OOP.

The properties of inheritance address reusability of software packages by building modules as extensions of existing ones. Inheritance allows the reuse and erfinement of abstract algorithms and data structures throughout the hierarchy of a system.

Another aspect of OOP, directed more towards extensibility, is the concept of polymorphism, and its natural complement, dynamic binding. (Dynamic binding has been formally shown in Chapter IV through the polymorphic assignment rule.)

Polymorphism means the ability to take several forms. In OOP, this refers to the ability of an entity to refer at run-time to instances of various classes (i.e., the same function name may bind to different codes when applied to different objects at run time.)

Dynamic binding is implemented in OOP by permitting the redefinition of a class operation in a descendant, and by having deferred operations whose implementation is only given in the descendants. For example, a polymorphic list called *aos_list* in the PCG served as an object stack (see Figure 13.) In Figure 13 a circle that is divided by a line represents an object. The name in the upper part of the circle represents the class type of the object; the name in the lower part of the circle represents the identifier of the object. The procedure *in* of the class List, which inserts an element into a list, has the following prototype:

$$\mathbf{P}_{L}^{in}(x)$$

where x is a variable and $D\_class(x) = PRIMITIVE$.

According to the polymorphic assignment rule, x could be assigned to any value $<o\ C\ n>$, $iff\ C \subset PRIMITIVE$; therefore, instances of the classes MI, MO, and VAR can be

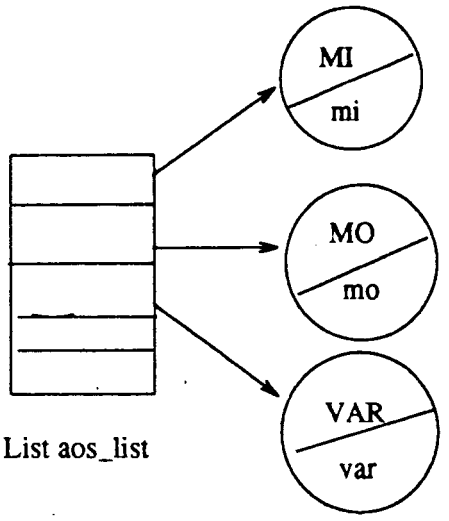


Figure 13. A polymorphic list as an AOS.

inserted into the *aos_list*. This example shows the use of polymorphism and dynamic binding in TAG90. By the same principle of polymorphism and dynamic binding, system extension of ADL can be implemented.

The generation and manipulation of a single object in the PCG has three steps:

1. An object is constructed;

2. The object is put into a stack called *aos_list*;

3. It is fetched from the stack and manipulated.

Assume that the methods for the *aos_list* are the procedure *put* and the function *get*. The virtual method defined in class PRIMITIVE for manipulating an instance of Family(PRIMITIVE) is *manipulate* and *manipulate* is also included in each member of the Family(PRIMITIVE). Assuming $D\_class(x) = PRIMITIVE$ and p is the list of primitive objects needed for the generation of x, the code notation for the an object generation and manipulation process is:

step 1: $x := \mathbf{S}_C^+ (p)$

step 2: *put*(x)*_aos_list*

step 3: $x :=$ *get*$()$ *_ aos_list*

manipulate$()\_x$

At step 1, an object variable x is bound to a C-object, where $C \subset PRIMITIVE$. This is a legal assignment according to the polymorphic assignment rule.

At step 2, x, which represents a set of objects,

$$x = \{ o \mid o = <i\ C\ n> \text{ and } C \in Family(PRIMITIVE) \}$$

is stored into the *aos_list.*

At step 3, x is fetched from the *aos_list* and manipulated. The transformation *manipulate()_x* is interpreted as the following semantic notation according to the polymorphic transformation rule:

$$\mathbf{P}_C^{manipulate}()\_<i\ C\ n> = <i\ C\ n'> \ iff\ S \in Family(PRIMITIVE).$$

The ADL object manipulation process at step 3 is polymorphic. The manipulation operation is dynamically bound to different implementations; the exact manipulation method depends on the actual type of x. The code for ADL object manipulation will not be changed if new ADL classes are added as the descendants of Class PRIMITIVE.

5.3 USER CONTROLLED EXTENSION OF ADL

5.3.1 ADL System Library (ASL)

Relevant research on organizing digital devices into an object-oriented data base can be found in [Wolf 1986; Ayer 1989; Gupt 1989]; however, none of these is incorporated into a high-level synthesis system. Since ASL is the knowledge base of TAG90, the user controlled extension of ADL adds adding knowledge into the ASL through the user interface with the ASL.

A DDS contains digital resources and an ADL program describes the resource elements and connections between them. There exist two kinds of resource elements: *storage elements* and *function elements*. An atomic ADL operation is described as:

$$<F\ OP_1\ OP_2 ..., OP_n>$$

where

- F is the function element of the operation.
- OP_i ($1 \leq i \leq n$)) is storage resource which is the input of F.

In current ADL F must be one of +, -, *, / operator; OP_1 and OP_2 must be one of the variables declared in the declaration section of ADL program. In a practical application F, OP_1, and OP_2 can be any type of variables, blocks, or structures. With the improvement of VLSI technology, new storage elements and functional units are being developed. There are sophisticated ADL user-defined systems and blocks that are very useful and should be made reusable. ASL serves as the warehouse of these new elements.

5.3.2 The ASL Element

ASL items are divided into different items and are implemented with C++ objects. There are currently two basic ASL items: *function* and *resource*. Each *item* contains one or more elements, which are the instances of the item. There is a wealth of information available, for each element in the data library, which could include size, aspect ratio, power, pin count and pin definitions, delays, input impedance, output drive, clocking requirements, and functions performed [McFa 1990-b]. User defined attributes can also be attached to each element. The way the ASL is organized is shown in Figure 14. In Figure 14 the ASL elements are organized under class ITEM (i.e., the ASL is the ITEM Family in terms of object-oriented design.)

The two basic class types in are:

- F_Elem, the ancestor class for function element classes.
- R_Elem, it is the ancestor class for resource element classes.

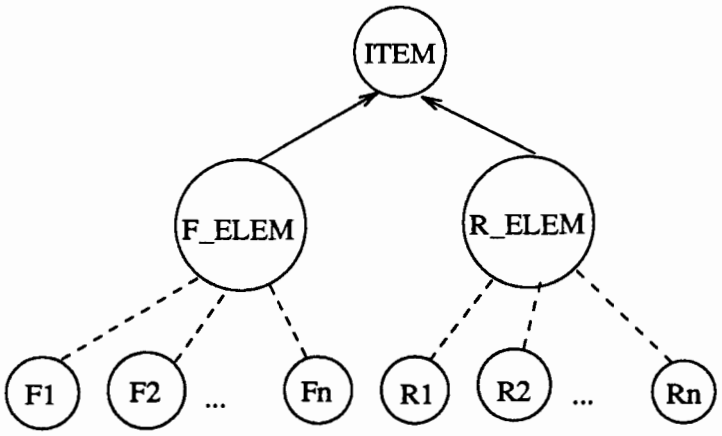


Figure 14. The class hierarchy of the ASL.

A pseudo class LIB_ELEM, which is assumed to be an element of the ASL, was mentioned in chapter IV. LIB_ELEM is either F_Elem or R_Elem; therefore, both class F_ELEM and class R_ELEM are supervised by the ADL classes PRIMITIVE and SYSTEM. This supervision relationship gives all members of both Family(PRIMITIVE) and Family(SYSTEM) the full accessibility to the ASL.

The instances of class F_Elem are different types of function elements: $F_1, F_2, .. , F_n$. The category of different function elements is determined by the user according to the technical needs. Basically, different function elements have different data operation features. The diagram of class F_ELEM is:

```
{
  class:
        F_ELEM
  inherit_from:
        ITEM
  supervised_by:
```

```
        SYSTEM, PRIMITIVE
    attributes:
        (
        name,
                        ;mnemonic of F
        op_number,
                        ;number of operation parameter
        op_type,
                        ;type array of the parameter array
        return_type,
                        ;type of the return value
        exec_time,
                        ;average execution time of the unit
        )
    method:
        (
        implementation,
                        ;method for hardware implementation
        check_usage
                        ;method for checking the legal usage of
                        ;the function element.
}
```

It should be pointed out that the diagram of F_ELEM is only an abstraction. It is convenient to specialize or modify this diagram by defining *subclass of F_ELEM*. An F_ELEM-object F_i represents conventional function calls in ADL -- in fact, any ADL statement can be represented in such a function call. For example:

```
a = add (x, y);  // represents a = x + y;
a = sin (x)  ;
a = a_block(x, y, z)
                // represents the call to a block called 'a_block'.
```

In the ADL program, if a *block* is called in the algorithm and the *block* is not defined, it may be an ASL function element; hence, TAG90 should be able to look up the called *block* in the ASL and verify that it exists and is appropriately used. On the other hand, the ASL user should be able to extend ADL so that new technologies and mature design methodologies could be easily incorporated to the DIADES system.

5.3.3 Adding New F_ELEM-object into ASL

Whenever a new ASL element is defined, new ADL statements become legal and the ASL, as well as TAG90, need to be extended. In the ASL all instances of F_ELEM family are stored in one hash table called *F_Hash_Table*. To add a new F_ELEM-object to ASL, the object is stored in F_Hash_Table. For this purpose, a separate file called *ADL Extension Format (AEF)* is defined to process the addition of a new F_ELEM-object. An AEF consists of user defined macros with the format of the macro being:

F_ext(name, op_number, op_type, return_type, time, implm)

where

- *name* is a character string representing the mnemonic of the new function element.
- *op_number* is an integer representing the number of parameters of the new function element.
- *op_type* is an integer vector indicating the types of the parameters. The length of the vector is equal to *op_number*. In ADL 1 indicates type *int;* 2 indicates type *float;* 3 indicates type *logical;* and *u_x* indicates a user defined type, where *x* can be a string. In fact, *u_x* is an instance of class R_ELEM.
- *return_type* is an integer indicating the type of the return value of the new function element.
- *time* is an integer indicating the average execution time of the function element.
- *implm* is a pointer to the function that describes the implementation details of the new function element.

To illustrate, when a new block *sin* is added to the ASL the user macro is:

F_ext("sin", 1, (2), 2, 5, f_sin);

The macros are transformed by the C preprocessor into F_ELEM-object constructors.

When the AEF and the main program are compiled and linked together, the ASL F_Hash_Table is built. The instances of F_ELEM are constructed and put into the F_Hash_Table. In an OOP language, an object is able to control its own behavior, such as "put itself into a hash table" by manipulating the pointer *this*, which points to the object itself at run time. After the AEF and the TAG90 main program are compiled together, the new TAG90, which is able to interpret the new function elements, is formed. All the recompilation process can be performed by a comprehensive *makefile* that does the recompilation and the link, leaving the least amount of work to the user. The extension process is shown in Figure 15.

5.3.4 Using ASL during ADL Analyzing Process

When an ADL program is compiled, the functional elements in the source ADL program are frequently checked to see if they are defined in the ASL. This process is called *ASL checking*. Class F_ELEM method *check_usage* is the function of class F_ELEM that checks the legal usage of an F_ELEM element. The instances of F_ELEM are stored in F_Hash_Table by their mnemonic names. An F_ELEM object can be found from the F_Hash_Table by its name and then the F_ELEM method check_usage is applied to the instance to verify if it is correctly used. The grammar rule for the function named *f_name* is:

```
f1
    : return_v ':=' f_name '(' parameter_list ')'
          { check_f($1, $3, $5); }
    |error
    ;
```

The ASL checking procedure *check_f* for this rule works using the following steps:

1. Find out from the F_Hash_Table the F_ELEM instance that has the name *f_name*.

2. If the instance is not found, report an error of calling an undefined function. Otherwise do 3.

3. Apply F_ELEM::check() function to the instance that is found, using $1, $3, $5 as parameters. If F_ELEM::check() returns 1, the usage of this function element is correct and the current statement is passed; otherwise, report an error of incorrectly using the F_ELEM statement.

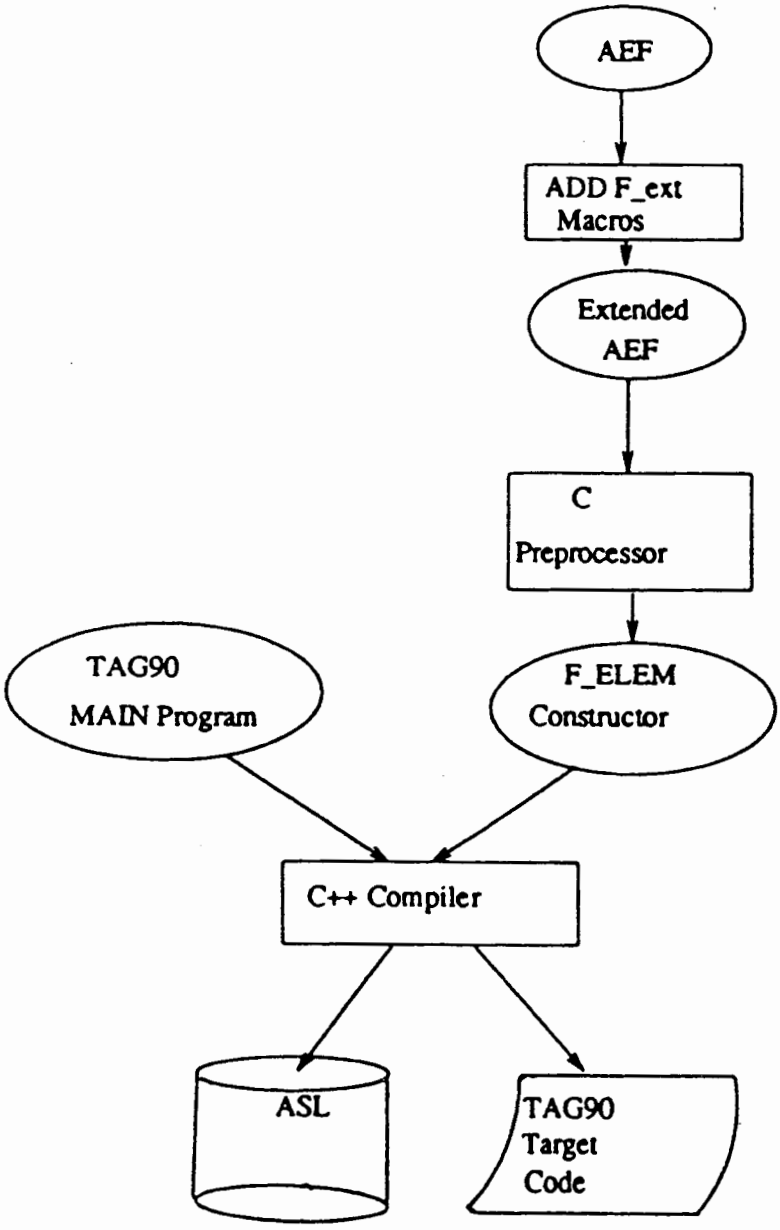


Figure 15. The user-controlled extension process of the ADL language

CHAPTER VI

THE PROGRAMMING ASPECTS OF TAG90

6.1 INTRODUCTION

The architecture of TAG90 program was presented in Chapter II. Some programming details will be discussed in detail in this chapter. TAG90 semantic routines (i.e., the PCG) are written in the C++ programming language [Wien 1988]. As mentioned in Chapter IV, PCG contains the class DAG and the *object manager*. The abstract diagrams of the classes was discussed in Chapter IV. The C++ code of the class DAG is listed in Appendix B. Some programming issues related to the design of the object manager, a composite of PCG, will be addressed in this chapter. This sections begins with some basic programming techniques for YACC and LEX. People who are familiar with YACC and LEX may pass over this part of discussion without losing any information essential to the main topic.

6.2 MORE ON THE ADL PARSER

The purpose of discussing some of the programming issues on the YACC grammar rules and the semantic routines is to aid those readers who are interested in the programming details of the TAG90 design.

6.2.1 YACC Value Stack

YACC value stack is a stack holding the values returned from the lexical analyzer and the semantic routines. In a LALR(1) grammar parsing, a *configuration* is written as following [Barr 1979]:

(state, stack, input-list)

A move is defined as a transformation of one configuration into another. There are two kinds of moves: *shift* and *reduce* when parsing. A shift means shifting terminal symbols from the *input list* into the *stack* top. A reduce means taking a handle w on the stack top and reducing it to a nonterminal symbol A for every production $A \rightarrow w$. When a shift takes place, the external variable *yylval* is copied onto the value stack. The pseudo-variables $$, $1, $2, etc. refer to the value stack. The semantics of the pseudo-variables are:

- $$ -- to return a value, the action normally sets "$$" to some value. In the above example "$$" is set to 1, which means the value of the stack is 1 if the type is *int*.
- $n (n is an integer greater than 1) -- to obtain the values returned by previous actions and the lexical analyzer, the action uses $1, $2, .. to refer to the values returned by the components of the right side of a rule, reading from left to right. Thus, if the rule is

  A : B C D;

  for example, $1 has the value returned by B and $3 the value returned by D.

The YACC value stack is an important component of the parser in that it facilitates an easy communication between the actions and the parser. For example,

```
go_stat
    : GO label
          { gen_goto($2); }
    ;
```

is an actual ADL grammar rule in the ADL parser. It recognizes a *goto* statement and

generates a *jump* MI according to the value of *label*. In this case $2 refers to the value of the nonterminal symbol "label," which is returned from the lexical analyzer, representing a label in the ADL program.

yylval, yyval, and the value stack can be used to hold a large variety of information. By default the value stack consists of *int* elements only. It can also be typed within a YACC specification. In fact, the YACC value stack is declared to be a *union* of the various types of values desired by the user. The user defines the union and associates the union member names to each token and nonterminal symbol having a value. When the value is referenced through a $$ or $n construction, YACC will automatically insert the appropriate union name. To declare the union, the user includes in the declaration section:

```
%union {
    <body of union>
    }
```

The above structure declares the YACC value stack, and the external variables *yylval* and *yyval* to have the type equal to this union. For example, in ADL Parser the union type is defined as

```
%union {
        ADL_VAR * y_sym;
                        /* field y_sym is a pointer
                           to a class ADL_VAR object */
        char    * y_str;
                        /* field y_str is a character string */
        int     y_count;
                        /* field y_count is an integer */
        int     y_type;
                        /* field y_type  is an integer */
        int     y_qualifier;
                        /* field y_qulifier is an integer */
    }
```

where

- *y_sym* has type ADL_VAR. ADL_VAR is defined as a class type, which is the C++ implementation for class VAR.

- *y_str* has a character string type.
- *y_count*, *y_type*, and *y_qualifier* have an *int* type.

Some tokens and nonterminals are associated with this union. For example,

```
%token <y_str> NAME
%token <y_type> INT
%token <y_type> LOGICAL
```

associates token *NAME* to type *y_str*, *INT* and *LOGICAL* to type *y_type*.

```
%type <y_type> type
%type <y_str> label
```

associates nonterminals *type* to *y_type*, and *label to y_str*.

6.2.2 Error Handling

The set of objectives of a satisfactory error recovery system in a compiler are as follows [Barr 1979]:

1. Report the error and indicate its location.
2. Diagnose the error as an aid to its correction.
3. Recover from the error so that the subsequent errors are detected. In TAG90 parser a function yywhere() is written to meet 1, 2, and 3.

There are three major classes of errors, *scanner, syntax* and *semantic* [Barr 1979].

The basic technique for handling syntactical errors is to treat an input error as a special case of a terminal symbol. For example, a simple expression containing only "-" operator will deal with this expression error. It is represented by the following YACC grammar rules:

```
expression
    : expression '-' expression
    | IDENTIFIER
    | NUMBER
    | error
    ;
```

When an error is encountered such as $x \ \& \ y$, where & is a unknown operator, the error will be reported to the user. The error will not cause the parser to exit; however, the parser will continue parsing the rest of the source program.

Semantic errors can only be detected by *the action routines*. In ADL program the following semantic errors are reported by TAG90:

- variable violation - most ADL programming errors have to do with variable errors. They could be "nondefined variable used," "wrong variable type used," etc.
- *go* error - the usage of the *goto* statement is restricted. One cannot go into a loop from the outside; one cannot go to a label that does not exist, etc..
- arithmetic error - division by zero is the most common arithmetic error.

The detection of semantic errors is conducted by local testing code at error-prone syntactical constructs. For example, a common label error in the ADL program is using the same label before more than one statement. To detect such an error, whenever a labeled statement is first seen by the parser, the label is stored into a *label hash table*. If the current label is already in the hash table, an error of *reusing a label* is reported:

```
lab_statement
    : label
        { .....
          if LABEL_HashTable.find($1) != NULL
                                    //If 'label' was used
                                    //before, an error is
                                    //reported.
            yyerror("Reused Label :");
            printf("%s", $1);
          .....
          }
     ':'
    | com_statement
    ;
```

In the above code *yyerror(char* str)* is a C++ procedure that reports an error on the standard output.

6.3 ADL VARIABLE HANDLING

ADL is a strictly typed language. All variables are declared in the declaration section except those that are generated internally during decomposing complex expressions at compilation time. An ADL variable is a character string and is considered an identifier except for the reserved words. An identifier is recognized in the scanner. It is necessary to deal with those identifiers right after they are found by the scanner. For instance, in the LEX specification file, a procedure screen() is the action routine after an identifier is found. Procedure *screen()* calls a function *look_up()* to check for a variable violation, to store a new variable, or to treat the identifier as an undefined identifier. The diagram of procedure *look_up()* is shown in Figure 16. Different variable handling processes are performed depending on where the new variable is encountered. The context of the new variable is determined by the so called *context variable*.

Lexical Tie-ins. Some lexical decisions depend on context. For example, procedure *look_up()* of the scanner sees an identifier *x* and wants to decide whether it is being defined or used in order to decide whether to store it into the variable hash table or

to check variable violation at this point. A context variable called *declaration_section* is declared in the parser to let the scanner know if the declaration section of the ADL program has been parsed yet. If the declaration section of the program is being parsed, variable declaration_section is set to 1; otherwise it is set to 0.

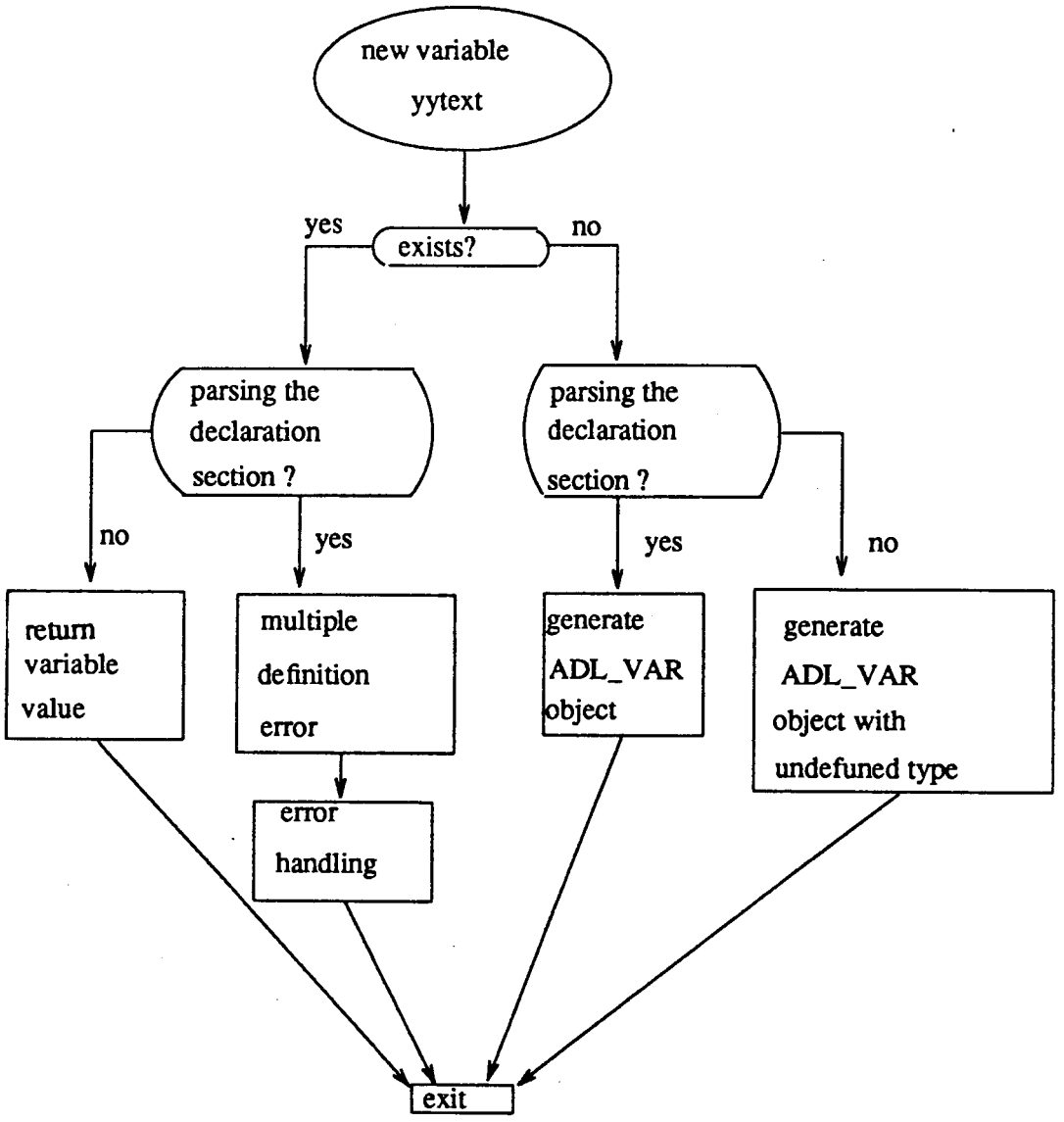


Figure 16. The diagram of the procedure look_up.

Since there are several context variables employed in the ADL parser, the danger is that it may violate the flexibility and the extensibility of the parser. To avoid this some

lexical decisions can be delayed to the syntactical decision level for the purpose of flexibility.

6.4 DESIGN OF THE PCG - THE GENERATION AND MANAGEMENT OF THE ADL OBJECTS

As mentioned in chapter IV, PCG consists of the ADL class DAG and the object manager. The implementation of the class DAG is the C++ definition of ADL classes. The implementation of the object manager is realized by calling the object constructor and various object transformation methods. Before addressing the issues of the construction and the management of objects, it is necessary to define ADL Object Stack.

[Definition6.1] ADL Object Stacks (AOS), are storage structures built for storing objects.

The generation and management of objects is carried out by the semantic routines of the ADL parser. After the parser recognizes ADL declarations or statements, one or more objects may be constructed and stored in the AOS. In TAG90 *hash table* and *generic list* are used as AOSes. For instance, VAR-objects are stored in a hash table; MI-objects are stored in a list.

The remainder of this chapter describes the details of constructing each class of object.

6.4.1 Constructing the Objects of the PRIMITIVE Family

The objects of the PRIMITIVE Family generated during the parsing process in the four steps previously described. The objects VAR, MI, and MO are constructed.

VAR Objects. There are two kinds of variables in an ADL program: the user defined variables and the compiler defined temporary variables. The user defined variables are in the declaration section; the temporary variables are generated when decomposing a complex expression.

The user defined VARs are generated in the scanner. Whenever an identifier is

seen by the scanner, a procedure *screen* is called to deal with it; a VAR object is generated if it is a new variable. In the scanner the rule of recognizing an identifier and the C code attached to it is as follows:

{letter}{letter_or_digit}* return screen();

Procedure screen() gets the character string of the identifier and stores it in a global variable called *yytext*. The *context variables* passed from the parser are *type_of_var* and *qualifier_of_rule*. The generated VAR are stored in the hash table called *var_hash*.

Compiler generated temporary VARs are generated when decomposing a complex expression such as:

$$a*(b - d)$$

A temporary variable named t_i is generated to hold the value of $b - d$. The name of the temporary variable starts with a *t*; *i* is incremented for each new temporary variable, insuring unique names.

The type of t_i is matched with the type of *b* and *d*; the *qualifier* of t_i is *TEMP*, which means temporary.

MI Objects. The three kinds of MI *seq* , *con*, and *jum*, represent *sequential, conditional*, and *jump* MIs. There are two kinds of statements in ADL: *assignment statements* and *control statements*. All sequential MIs are created from the assignment statements. The rule and its action for generating sequential MIs is:

```
assign_stat
   : lhs ':=' rhs
         {
          gen_mi();
         }
   | error
   ;
```

gen_mi() is the procedure constructing an instance of class MI and storing it into a list called *mi_list*, which is an instance of the generic list class *List*. The action of *gen_mi()* can be regarded as equal as the following semantic notations:

$$\mathbf{S}^{+}_{MI}\ (cv) = mi \quad \text{..... (1)}$$

$$\mathbf{P}^{insert}_{List}\ (<mi\ \ MI\ \ n>)\_<mi\_list\ \ List\ \ n1> = <mi\_list\ \ List\ \ n2> \quad \text{..... (2)}$$

In (1) a *MI-object mi* is synthesized from a set of context variable *cv*. In (2) the *mi* inserted into a list called *mi_list*.

con and *jum* MIs are created the control statements. The construction of MI instance from the control statement involves the modification of the values of feature *ne (next)* and feature *br (branch)* of an MI instance. For example, an if-else statement:

```
if ( a > 0 )        (1)
    answer := a  (2)
  else
    answer := b;  (3)
goto S;             (4)
```

will cause the generation of four MIs corresponding to the labeled ADL clause as listed above. When (1) is parsed, an instance of MI (an MI) is constructed (mi_1), but the branch address of this MI is unknown until (3) is parsed, then the *feature br* of mi_1 is modified to be the *MI id* of mi_3 (statement *answer := b*;). The semantic notation for feature modification of an MI instance follows:

$$\mathbf{P}^{modif}_{MI}\ (a_1, a_2)\_<mi\ \ MI\ \ n_1> = <mi\ \ MI\ \ n_2>,$$

where a_1 and a_2 are the attributes set of *mi* before and after the modification; *modif* is the modification procedure defined in MI.

MO Objects. The grammar rule for ADL expression is:

```
expr
    : expr op expr
        {
        $$ = gen_mo(get_mo_id(), $2, get_mi_id(), $1, $3);
                    // Function gen_mo() returns a VAR object
                    // which is a temporary destination output of
                    // the expression represented by the rule.
```

```
        }
    | term
        {
        $$ = yylval;  // An VAR object is generated in the
                      // scanner.
        }
    | '(' expr ')'
    | error
    ;

term
    : NAME
    | number
    ;

op
    : '+'
    | '-'
    | '*'
    ;
```

gen_mo() is the procedure constructing an instance of MO. The return value of *gen_mo()* is an instance of VAR, which is the destination element of an MO; *get_mo_id()* and *get_mi_id()* return the current number of MIs and MOs generated in the parsing process. The constructed MO object is inserted into a list called *mo_list.*

6.4.2 Constructing Objects of the SYSTEM Family

The SYSTEM-object *adl_system* is constructed before the parser starts. *adl_system* represents the abstract architecture of the target digital system and contains the constraints set by the user. The instances of other members in the Family(SYSTEM), DP, CU, and DDS are constructed after the parsing process. The constructions of CU, DP, and DDS will now be discussed.

CU Object. A complete DDS has at least one *Control Unit.* As mentioned before, the CU representation is in the MI-list format, which is similar to a *microprogram.* Every time an MI is generated, it is stored in a list called *mi_list.* Object *mi_list* is an instance of class *List.* When the parser comes to an end, the construction of the *mi_list* has been completed. The CU-object *adl_cu* is constructed from the *mi_list.* The

transformation formula for this is:

$$\mathbf{S}^{+}_{CU}\ (<mi\_list, List, n>) = <adl\_cu, CU, n'>$$

DP Objects. The DP is represented by the data flow format (i.e., the list of *MO-objects*.) The hash table of VAR objects is supplementary. VAR objects are put into a hash table called *var_hash*; MO objects are put into a list called *mo_list*; and object *mo_list* is an instance of class *LIST*. When the parser comes to an end, a DP-object called *adl_dp* is synthesised. The transformation formula for *adl_dp* construction is illustrated as:

$$\mathbf{S}^{+}_{DP}\ (<mo\_list, LIST, n1>, <var\_hash, Hash, n2>) = <adl\_dp, DP, n3>$$

DDS Object. Class DDS is an abstract class that has accessibility to all information of the objects *adl_dp* and *adl_cu*, since class DDS is the supervisor class of classes DP and CU. A function called *gen_p_graph* is defined as the only method of class DDS. Since the instance of DDS is a constant, the DDS-object called *adl_dds* is constructed before parsing the ADL program.

6.5 P-GRAPH GENERATION

The DDS object *adl_dds* invokes its function *gen_p_graph* in order to generate P-graph code after the objects *adl_cu* and *adl_dp* are generated; they encapsulate all information and most functions needed for the P-graph code generation. The DDS-object *adl_dds* supervises the DP-object *adl_dp* and the CU-object *adl_cu*. The function called *gen_p_graph* uses data and methods stored in adl_cu and *adl_dp* to generate an entire P-graph list. The C++ definition of *gen_p_graph* is:

```
DDS::gen_p_graph(DP d, CU c)
{

   c.gen_coplisset();
            //generate *coplisset*
   d.gen_nalisset();
            //generate *nalisset*
   d.gen_lzmset();
            //generate *lzmset*
   c.gen_nolisset();
            //generate *nolisset*
   gen_otherlist(c, d);
            //generate *plisset*

}
```

The C++ code for invoking function *gen_p_graph* is:

gen_p_graph(adl_dp,˜adl_cu).adl_dds

In this definition the method *gen_p_graph* accesses the attributes of the objects *CU* and *DP* and generates the following P-graph lists: **coplisset**, **nolisset**, **nalisset**, **plisset**, and **lzmset**.

CHAPTER VII

CONCLUSION AND FUTURE WORK

7.1 SUMMARY

In this thesis, the design of the ADL analyser TAG90 design process is presented. It shows that there are still many issues that need to be investigated in the area of the SLA design of a high-level synthesis system. Object-oriented programming and object-oriented design methods are used in this project. The advantages of using OOP, which the author had a chance to appreciate during the design process, are summarized in the following points:

- The complexity of TAG90 design has been tremendously reduced.
- The extensibility of TAG90 has been put into use in the stepwise refinement of the TAG90 itself. The simplicity of the extension of the TAG90 system is very impressive.
- The methodologies employed in the TAG90 design are expected to transcend the traditional top-down design strategies.

7.2 FUTURE WORK

TAG90 is the heart of the DIADES system. It employs *object oriented design and programming* techniques and has shown its advantages in some prospects, such as its extensibility and simplicity. Currently, TAG90 has been implemented with the AT&T C++ Language System Release 2.0. It is able to translate most ADL programs into P-graph and do the translation work lot faster than the old ADL translator. The parallel

programming *fork* and *joint* statement are also being incorporated. Some of the object transformations are yet to be implemented. The future of DIADES relies on how TAG90 is improved. Some further research and programming work in TAG90 need to be done:

- Adding OOP language features to ADL. So far, ADL is not an OOP language. If ADL is extended or even redefined as an OOP language, some new and creative features of the entire DIADES system will be realized. For example, the new ADL could include user defined class types.

- More understanding of the role of the objects. Currently the concept of the object is limited to that of the objects found in object-oriented programming languages, which package operations for data manipulation with data itself. To model the hardware design entities, a more sophisticated object model is expected to be established in the future. An object-oriented CAD data model is discussed in [Katz 1987]. More precise definition and implementation of *CAD objects* needs to be found.

- Adding new methods to ADL classes. Currently, the methods defined for the ADL classes are limited to the formating and data setting aspects. More sophisticated *procedures* and *function* related to the *optimization* and *communication* between objects are expected to emerge.

- Making more use of the ADL class DAG in the future. Currently, the ADL objects seem like yet another IR between ADL and P-graph. As a matter of fact, the ADL objects can have a much wider usage. It will eventually replace P-graph and become a new generation of IR for high-level synthesis system.

REFERENCES

[Aho 1989] A. V. Aho, R. Sethi, J. D. Ulman, "Compilers, principles, techniques, and tools," Addison-Wesley, 1989.

[Ayer 1989] K. E. Ayers, "An object-oriented logic simulator," *Dr. Dobb's Software Tools*, Vol. 14, No. 12, pp. 72-76, December, 1989.

[Aylo 1986] J. H. Aylor, R. Waxman and C. Scarratt, "VHDL -- Feature description and analysis," *IEEE Design and Test*, Vol. 3, No. 4, pp. 17-27, April, 1986.

[Barb 1981] M. R. Barbacci, "Instruction Set Processor Specifications (ISPS): the notation and its applications," *IEEE Trans. on Computers*, Vol. C-30, No.1, pp. 24-40, January, 1981.

[Barr 1979] William A. Barrett, "Compiler construction: Theory and practice," 1979, Science Research Associates, Inc.

[Camp 1989] R. Camposano and W. Rosenstiel, "Synthesizing circuits from behavioral description," *IEEE Trans. on CAD*, Vol. 8, No. 2, pp. 171-180, February, 1989.

[DeMa 1986] H. DeMan, J. Rabey, P. Six, and L. Claesen, "Cathedral_II: A silicon compiler for digital signal processing," *IEEE Design and Test*, Vol. 3, No. 6, pp. 13-25, December, 1986.

[Farr 1989] R. Farrow and A. G. Stanculescu, "A VHDL compiler based on attribute grammar methodology," *ACM SIGPLAN's 89 Conference on Programming Language Design and Implementation*, Portland, Oregon, pp. 120-130, June, 1989.

[Ghos 1988] S. Ghosh, "Using ADA as an HDL," *IEEE Design and Test*, Vol. 5, No. 1, pp. 30-42, February, 1988.

[Goos 1990] Gert Goossens, Jan Rabey, Joos Vandewalle, and Hugo De Man, "An efficient microcode compiler for application specific DSP Processors," *IEEE Trans. on CAD*, Vol. 9, No. 9, pp. 925-937, September, 1990

[Gupt 1989] R. Gupta, W. H. Chang, R. Gupta, I. Hardonag, and M. A. Breuer, "An object-oriented VLSI CAD framework," *IEEE Computer*, Vol. 22, No. 5, pp. 28-36, May, 1989.

[Hafe 1982] L. J. Hafer and A. C. Parker, "Automated synthesis of digital hardware," *IEEE Trans. on Computers*, Vol. C-31, No. 2, pp. 33-43, February, 1982.

[Katz 1987] Randy H. Katz, Rajiv Bhateja, Ellis E-Li Chang, David Gedye, and Vony Trijanto, "Design version management," *IEEE Design & Test*, Vol. 4, No. 1, pp. 12-22, February, 1987.

[Kim 1990] Won Kim, "Object-oriented database: Definition and research directions," *IEEE Trans. on Knowledge and Data Engineering*, Vol. 2, No. 3, pp. 327-341, September, 1990.

[Kowa 1985] T. J. Kowalski, "The VLSI design automation assistant: from algorithm to Silicon," *IEEE Design and Test*, Vol. 2, No. 4, pp. 33-43, August, 1985.

[Lips 1986] R. Lipsett, E. Marschner, and M. Shahdad, "VHDL -- The language," *IEEE Design and Test*, Vol. 3, No. 2, pp. 28-41, April, 1986.

[Liu 1989] J. Liu, "A finite state machine synthesizer," *Master Thesis*, the Department of Electrical Engineering, Portland State University, 1989.

[Marw 1979] P. Marwedel, "The MIMOLA design system: detailed description of the software system," in *Proceedings of the 16th Design Automation Conference*, New York, NY, pp. 59-63, June, 1979.

[McFa 1983] M. C. McFarland and A. C. Parker, "An abstract model of behavior for hardware descriptions," *IEEE Trans. on Computer*, Vol. c-32, No. 7, pp. 621-631, July, 1983.

[McFa 1990-a] M. C. McFarland, A.C. Parker, and R. Camposano, "The high-level synthesis of digital systems," *IEEE proceedings*, Vol. 78, No. 2, pp. 301-318, February, 1990.

[McFa 1990-b] M. C. McFarland and T. J. Kowalski, "Incorporating bottom-up design into hardware synthesis," *IEEE Trans. on CAD*, Vol. 9, No. 9, pp. 938-950, September, 1990.

[Meye 1988] Bertrand Meyer, "Object-oriented software construction," Prentice Hall, 1988.

[Mili 1990] H. Mili, J. Sibert and Y. Intrator, "An object-oriented model based on relations," *The Journal of Systems and Software*, Vol. 12, No. 2, pp. 139-155, May, 1990.

[Pang 1987] B. M. Pangrle and D. D. Gajski, "Design tools for intelligent silicon compilation," *IEEE Trans. on CAD*, Vol. CAD-6, No. 11, pp. 1098-1112, November, 1987.

[Park 1986] A. Parker, J. Pizarro, and M. Mlinon, "MAHA: A program for data path synthesis," in *Proc. of 23rd DAC.*, New York, NY, pp. 461-466, June, 1986.

[Park 1988] N. Park and A. C. Parker, "Sehwa: A software package for synthesis of digital hardware from behavioral specifications," *IEEE Trans. on CAD*, Vol. 7, No. 3, pp. 356-370, March, 1988.

[Paul 1986] P. G. Paulin and J. P. Knight, and E. F. Girczyc, "HAL: A multi-paradigm approach to automatic data path synthesis," in *Proceedings of the 23rd Design Automation Conference*, New York, NY, pp. 263-270, June, 1986.

[Paul 1989] P. G. Paulin and J. P. Knight, "Force directed scheduling for the behavioral synthesis of ASIC's," *IEEE Tran. on CAD*, Vol. 8, No. 6, pp. 661-679, June, 1989.

[Perk 1988] M. A. Perkowski, "Control unit design in DIADES," *technical report of the DIADES research group*, the Department of Electrical Engineering, Portland State University, July, 1988.

[Schr 1985] A.T. Schreiner and H.G. Friendman,Jr., "Introduction to compiler construction with Unix," Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1985.

[Smit 1988] D. Smith, "Data path design in the DIADES automation system," *technical report of the DIADES research group*, the Department of Electrical Engineering, Portland State University, July, 1988.

[Sout 1983] J. R. Southard, "MacPitts: An approach to silicon compilation," *Computer*, Vol. 16, No. 12, pp. 74-82, December, 1983.

[Sugi 1986] A. Sugimoto, S. ABE, M. Kuroda, and Y. Kato, "An object-oriented visual simulator for microprogram development," in *Proceedings of 23th Design Automation Conference*, New York, NY, pp. 138-144, June, 1986.

[Sun 1988] L.-F. Sun, J.-M. Liaw, and T. -M. Parng, "Automated synthesis of microprogrammed controllers in digital systems," *IEE Proceedings Computers and Digital Techniques*, Vol. 135, Part E, No. 4, pp. 23-30, July, 1988.

[Tric 1985] H. W. Trickey, "Compiling Pascal programs into silicon," *Ph.D. dissertation*, Standford University, July, 1985.

[Tric 1987] H. W. Trickey, "Flamel: A high-level hardware compiler," *IEEE Trans. on CAD*, Vol. CAD-6, No. 2, pp. 256-269, March, 1987.

[Tsen 1986] Tseng and D. P. Siewiorek, "Automated synthesis of data paths in digital systems," *IEEE Trans. on CAD*, Vol. CAD-4, No. 7, pp. 379-395, July, 1986.

[Wien 1988] R. S. Wiener, L. J. Pinson, "An introduction to object-oriented programming and C++," Addison-Wesley Publishing Company, 1988.

[Wils 1990] P. A. Wilsey and Subrata Dasgupta, "A formal model of computer architectures for digital system design environments," *IEEE Trans. on CAD*, Vol. 9. No. 5, May, 1990.

[Wils 1987] P. A. Wilsey, "A hardware description language for multilevel design automation systems," *Ph.D. dissertation*, Univ. of Southwest Louisiana, Lafayette, LA, 1987.

[Wolf 1986] W. H. Wolf, "Fred: A procedural data base for VLSI design," in *Proceedings of 23th Design Automation Conference*, New York, NY, pp. 473-486, June, 1986.

[Yang 1988] L. Yang and M. A. Perkowski,"Automated synthesis of microprogrammed control unit in the DIADES system," *technical report of the DIADES research group*, the Department of Electrical Engineering, Portland State University, November, 1988.

APPENDIX A

A COMPLETE EXAMPLE OF RUNNING TAG90

APPENDIX A

A COMPLETE EXAMPLE OF RUNNING TAG90

An Example ADL program (in file adl.ex):

----------adl.ex----------

```
adl a example_circuit ;
                                /* Program Heaad */
/* Declaration Section: */
input { int op1, op2;}
                                /* Input Variables */
intern { logical temp; }
                                /* Internal Variables */
output { int answer; }
                                /* Output Variables */
/* Algorithm Section: */
start;
 l0: temp := op1 + op2;
    if (temp == 10)
      temp := op1 - op2;
    answer := temp;
    go l0;
end.
```

The following is the typescript file of running TAG90:

----------typescript----------

Script started on Sun Sep 23 00:19:15 1990

[lian][/usr/export/home/jove/hal/lian] TAG90 adl.ex

Ending declaration, the ADL_VAR_HashTable is:

(op1, 1, 1, 0)

(op2, 1, 1, 0)

(answer, 1, 2, 0)

(temp, 2, 3, 0)

Ending compiler,

the control flow (ADL_MI_List) is:

(1 seq 2 0 (1) -)

(2 con 3 4 (2) -)

(3 seq 4 0 (3) -)

(4 jum 1 0 (4) -)

(6 end 0 0 ( ) -)

the data flow (ADL_MO_List) is:

(1, +, 1, (op1, op2), temp)

(2, ==, 2, (temp, 10), ALU_com)

(3, -, 3, (op1, op2), temp)

(4, (), 4, (temp), answer)

Label list (ADL_Label_List):

```
( 10 1 )

Goto list:

Generate P-graph representation ?
Y

P-graph representation for adl.ex:

List Of Arrows:

 (coplisset
 (1 (x 1 2)
   (2 2 3)
   ((not 2) 2 4)
   (x 3 4)
   (x 4 1)))

List Of Nodes:

(nolisset
 (1 (3 3 nil)
   (4 4 nil)
   (cond 2 nil)
   (start 1 nil)))

List Of Descriptions:

(nalisset
 (1 (1 (:= temp (plus op1 op2)))
   (3 (:= temp (sub op1 op2)))
```

```
   (4 (:= answer temp))))

List Of Predicates:

(plisset
 (1 (2 (equal temp 10))))

List Of Variables:

(lzmset
 (1 (op1)
   (op2)
   (temp)
   (answer)))
----------end of typescript ----------
```

Note:

In above exapmle, an ADL program, *adl.ex,* is analysed. During parsing adl.ex, the VAR objects are generated and stored in the *var_table*. The MI objects are generated and stored in the *mi_list*. The MO objects are generated and stored in the *mo_list*. After parsing the *declaration section* the var_table is formed and printed. After the entire parsing process, *mo_list* and *mi_list* are formed and printed. The P-graph is generated.

ADL_Label_List and *ADL_Go_List* are internal lists used in TAG90 program. They are also listed for debugging purpose.

APPENDIX B

YACC SPECIFICATION FILE OF THE ADL

APPENDIX B

YACC SPECIFICATION FILE OF THE ADL

```
/* The declaration section: */

%{
#include "adl.h"
#include "instrct.h"
#define ERROR(x)  yywhere(), puts(x)

extern char yytext[];
extern void mod_branch(int, int);
                                    /* procedure for modifying the branch address of
                                       a micro-instruction (MI) */
extern void mod_next(int, int);
                                    /* procedure for modifying the 'next' address of
                                       a micro-instruction (MI) */
extern void print_instrct();

                                    /* old procedure for print all MI in the MI-list.
                                       Now it is a method of class ADL_CU (control unit). */
extern void gen_goto(char*);
                                    /* procedure for generating a goto MI. */
extern void gen_goto1();
extern void yyerror(register char* );
                                    /* procedure for producing an error message. */
extern void yywhere();
                                    /* procedure for locating an error in the source program. */
extern void gen_instrct(int , int, int, int, ADL_OPERATION*);

                                    /* procedure for generating a MI. The MI generated here
                                       might be temporary since its 'branch address' or 'next
                                       address' need to be modified. */
extern void gen_cond();
                                    /* procedure for generating a conditional MI. */
extern int yylex();
                                    /* the lexical analyser (scanner) function */

Hash VAR_HashTable(VarHashTableLen);
                                    /* hash table for variables */
Hash LABEL_HashTable(LabelHashTableLen);
                                    /* hash table for labels */
Hash go_stack(LabelHashTableLen);
```

```
                                        /* hash table for goto statement */
ADL_INSTRCT_ID adl_instrct_id;
                                        /* counter thats counts the number of the current MI */
List adl_instrct_list;
                                        /* List that stores ADL internal instruction */
                                        /* generated during parsing. */
ADL_INSTRCT *instrct_pointer ;
                                        /* ADL_INSTRCT is class type for MI objects */
Var_type        type_of_rule;
Var_qualifier   qualifier_of_rule;
int             declaration_section;
                                        /* context variable marking the declaration section */
%}
%union {
        ADL_VAR     * y_sym;
                                        /* Identifier */
        char *  y_str;
                                        /* Constant */
        int       y_count;
                                        /* Count */
        int       y_label;
                                        /* Label */
        int     y_type;
                                        /* Variable type */
        int     y_qualifier;
    }

                                        /*
                                         *      terminal symbols
                                         */
%token <y_str> CONSTANT
                                        /* ADL number constant */
%token <y_sym> NAME
                                        /* variables and other identities */
%token <y_type> INT
                                        /* reserved word 'int' */
%token <y_type> LOGICAL
                                        /* reserved word 'logical' */
%token ADL
                                        /* reserved word */
%token AND
                                        /* reserved word 'and' */
%token BLOCK
                                        /* reserved word 'block' */
%token ELSE
```

```
%token ELSE          /* reserved word 'else' */
%token END           /* reserved word 'end' */
%token EQ            /* operator '==' */
%token EXOR          /* reserved word 'exor' */
%token FOR           /* reserved word 'for' */
%token GR            /* operator '>'  */
%token GEQ           /* operator '>='  */
%token GO            /* reserved word 'go' */
%token IF            /* reserved word 'if' */
%token INPUT         /* reserved word 'input' */
%token INTERN        /* reserved word 'intern' */
%token LE            /* assignment operator ':=' */
%token LEQ           /* operator '<=' */
%token LESS          /* operator '<' */
%token NAND          /* reserved word 'nand' */
%token NOR           /* reserved word 'nor' */
%token NOT           /* reserved word 'not' */
%token OR            /* reserved word 'or' */
%token OUTPUT        /* reserved word 'output' */
%token RE
%token RESET         /* reserved word 'reset' */
%token SET           /* reserved word 'set' */
%token SIM           /* reserved word 'sim' */
%token START         /* reserved word 'start' */
%token WAIT          /* reserved word 'wait' */
%token WHILE         /* reserved word 'while' */
```

```
                                        /*
                                         *      typed non-terminal symbols
                                         */

%type  <y_type> type
%type  <y_label> if_prefix
%type  <y_str>  label


                                        /*
                                         *    precedence table
                                         */

%left GR LESS GEQ LEQ
%right '!'
                                        /* NOT */
%left '@' '^' '$'
%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'

%%

/* The rules section */

/* An ADL program: */

program
        : program_head program_body
                { yyerrok; }
        | error
        | program_head error
        ;

/* ADL program head part: */

program_head
        : ADL program_id program_name ';'
                { yyerrok; }
        | error
        ;
```

```
/* the identifier of an ADL program: */

program_id
        : NAME
        ;

/* the user-chosen name of an ADL program: */

program_name
        : NAME
        ;

/* the body of an ADL program (usually including two parts: */

program_body
        : declarations
                { printf("Ending declaration: print Hash values : 0);
                  VAR_HashTable.printHash();
                  declaration_section = 0;
                  yyerrok; }
          code
        | error
        | declarations error
        | error code
        ;

declarations
        :       { declaration_section = 1; }
                                        /*Tell lex that parser is parsing the source's
                                          declaration section */
          input_var intern_var output_var subroutines
        | intern_var
                                        /*Intern variables suffices to form a DDS .*/
        | error
        ;

input_var
        :       { qualifier_of_rule = 1; }
          INPUT '{' var_declarations rr
        | error
        ;

intern_var
        :       { qualifier_of_rule = 3; }
          INTERN '{' var_declarations rr
        | error
        ;

subroutines
        :
                                        /* Can be empty: this part is optional */
        | block_declaration ;
```

```
block_declaration
                                        /* Declare a block */
        : BLOCK '{' blocks sc rr
        ;


blocks
        : blocks ',' block
                                        /* More than one block can be declared
                                           in a block decl section */
        | block
        ;


block
        :
                                        /* null */
                                        /* Block looks like a function. */
        | block_name '(' arguments ')'
        ;


block_name
        : NAME
                { ; }
        ;


arguments
                                        /* Block arguments: list of variables. */
        :
                                        /* Can Be Empty */
        | arguments ',' argument
                { yyerrok; }
        | error
        | argument
        ;


argument
                                        /* Each block argument is an ADL variable */
        : NAME
        ;


output_var
        :       { qualifier_of_rule = 2; }
                                        /* context variable for the lexical analyser. */
          OUTPUT '{' var_declarations rr
        | error
        ;
```

```
var_declarations
                                        /* A general rule for all types of variable decl. . */
        :
                                        /* An empty variable is acceptable */
        | var_declarations  var_declaration
        | var_declarations error
        ;

/* ADL variable declaration section: */

var_declaration
        : type
        { type_of_rule = $1; }
          var_list sc
        ;

/* ADL variable list in the declaration section */

var_list
        : NAME
                { ; }
        | var_list ',' NAME
                { yyerrok; }
        | var_list error
        | var_list error NAME
                { yyerrok; }
        | var_list ',' error
        | NAME '[' number ']'
                                        /* Variable array declaration. */
        | '[' number ']' NAME
        | '[' number ']' NAME '[' number ']'
        ;


type
                                        /* Only two types in ADL so far */
        : INT
                { $$ = 1; }
        | LOGICAL
                { $$ = 2; }
        ;

number
        : CONSTANT
                { ; }
        ;

code
                                        /* The algorithmic section of an ADL program. */
        : START  statements END
                {
                 getList get_aiList(&adl_instrct_list);
```

```
                                        /* getList of List adl_instrct_list */
                gen_instrct(adl_instrct_id.getIid(), 0, 0, 4, 0);

                                        /* Generate an 'end MI'. */
            gen_goto1();
                print_instrct( );
                                        /* Print the list of MI. */
                printf("Label list: \n");
                LABEL_HashTable.printHash();

                                        /* Print all labels used in the program and their locations */
            printf("Goto list: 0);
            go_stack.printHash();
                yyerrok;
              }
        | error
        ;

statements
        : statements  statement
        | statement
        ;

statement
        :
                                        /* null */
        | com_statement
                                        /* common statement */
        | lab_statement
                                        /* labeled statement */

com_statement
        : assign_stat sc
                                        /* Assignment statement */
        | go_stat sc
                                        /* go statement  */
        | if_statement
                                        /* if statement  */
        | while_stat
                                        /* while statement */
        | for_stat
                                        /* for statement */
        | wait_stat
                                        /* wait statement */
        | SET NAME
                                        /* set statement  */
        | RESET NAME
                                        /* reset statement */
        | SIM '{' assign_stats '}'
                                        /* simultaneous statement for parallel execution. */
        | error
                { ERROR("statement error"); }
        ;
```

```
go_stat
        : GO label
                { gen_goto($2); }
        ;

lab_statement
        : label
                                        /* special treatment is needed for a labeled
                                           statement: */
                { ADL_LABEL *lp = new ADL_LABEL($1, 0);

                 if ((ADL_LABEL*)LABEL_HashTable.find($1) == NULL)

                                        /* test if the label here has been used before. */

                                        /* if not, generate a new MI and register the label. */

            {
                  int i = adl_instrct_id.getIid();
                  lp -> putAddress(i);
                  LABEL_HashTable.insert(lp);
                 }
                else
            {
                  yyerror("Reused label:");
             printf("%s \n", $1);
                 }
               }
          ':'
        | com_statement
        ;

/* the label of an ADL statement: */

label
        : NAME
                { $$ = $1 -> getId(); }
        | CONSTANT
                { $$ = $1; }
        ;

/* the set of assignment statements: */

assign_stats
        : assign_stats assign_stat
        | assign_stat
        ;

/* ADL assignment statement: */
assign_stat
        : lhs ass_op rhs
                { int i = adl_instrct_id.getIid(),
                   n = i + 1,   b, t = SEQ;
```

```
                gen_instrct(i, n, b, t, 0);
            adl_instrct_id.addIid();
                }
        | error
        ;

/* ADL assignment operator: */

ass_op
        : RE
        ;

/* the left side of an ADL expression: */

lhs
        : NAME
                {
                  if($1 -> getType() == UNDEC)
                   {
                    yyerror("Undeclared variable");
                  printf("%s 0, $1 -> getId());
                   }
                }
        | NAME '@' NAME
        ;

rhs
        : a_expr
                                        /* arithmetic expression */
        | l_expr
                                        /* logical expression */
        ;

/* arithmetic expression: */

a_expr
        : a_expr '+' term
        | a_expr '-' term
        | NAME RE  a_expr
        | a_expr EQ  a_expr
        | a_expr GR  a_expr
        | a_expr GEQ a_expr
        | NAME LE  a_expr
        | a_expr LESS a_expr
        | '(' a_expr ')'
        | term
        | error
        ;

term
        : term '*' unary
        | term '/' unary
```

```
        | term '%' unary
        | term '|' unary
        | term '&' unary
        | term '@' unary
        | unary
        ;

unary
        : primary
        | '-' primary
        ;

primary
        : NAME
                                        /* variable */
        | number
                                        /* constant */
        | b_call
                                        /* block call */
        | bit_range
                                        /* bit range operation */
        | index
                                        /* array operation */
        ;

/* bit range of a parallel variable: */

bit_range
        : NAME '@' number '@' number;
        ;

/* block call */

b_call
        : NAME '(' parameters ')'
        ;

/* block parameter: */

parameters
        :
                                        /* Can be empty */
        | parameters ',' parameter
        | parameter
        ;

parameter
        : NAME
        ;


/* bit position of a variable: */
```

```
index
        : NAME '[' NAME ']'
        | NAME '[' number ']'
        ;

/* logic expression: */

l_expr
        :
        | '(' l_op l_expr l_expr ')'
        | NAME
        | bit
        ;

/* logic operator: */

l_op
        : AND
        | OR
        | NAND
        | NOT
        ;

bit
        : '0'
        | '1'
        ;

/* rule for "if statement": */

if_statement
        : if_prefix compound_statement
                {
                 mod_branch($1, adl_instrct_id.getIid());
                }
        | if_prefix compound_statement ELSE
                {
                $<y_label>$ = adl_instrct_id.getIid() - 1;
                 mod_branch($1, adl_instrct_id.getIid());
                }
          compound_statement
                {
                 mod_next($<y_label>4, adl_instrct_id.getIid());
                }
        ;

if_prefix
        : IF '(' rhs rp
                {
                 $$ = adl_instrct_id.getIid();
                 gen_cond();
                }
        ;
```

```
/* rule for "while-loop", syntax like that in C language: */

while_stat
        : WHILE '(' rhs rp
                { int i = adl_instrct_id.getIid(),
                    n = i + 1;
                 gen_instrct(i, n, 0, 2, 0);
                 $<y_label>$ = i;
                 adl_instrct_id.addIid();
                }
          compound_statement
                {
                 int i = adl_instrct_id.getIid();
                 gen_instrct(i, $<y_label>5, 0, 3, 0);
            adl_instrct_id.addIid();
                 mod_branch($<y_label>5, i+1);
                }
        ;

/* rule for "for-loop", syntax like that in C language */

for_stat
        : FOR '(' for_initstmt sc rhs sc rhs rp compound_statement
        ;

for_initstmt
        :
                                        /* maybe empty */
        | assign_stat
        ;

/* a group of statements: */

compound_statement
        : com_statement
        | '{' list_stmt rr

/* a list of statements: */

list_stmt
        : list_stmt com_statement
        | com_statement
        ;

/* rule for "wait statement: */

wait_stat
        : WAIT number sc
        | WAIT WHILE '(' rhs rp sc
        ;
```

```
                                        /*
                                         * make certain terminal symbols very important
                                         */

rp      : ')'  { yyerrok; }
sc      : ';'  { yyerrok; }
rr      : '}'  { yyerrok; }

%%
```